从 $e^{i\pi}+1$ 开始

用Python做科学计算

HYRY Studio 著

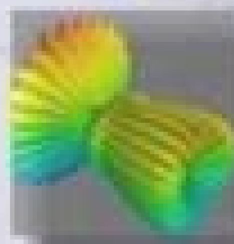
NumPy SciPy Matplotlib Chaco TraitsUI VTK Mayavi Sympy

数字信号 滤波器 频域处理 声音 图像 数据可视化 动画模拟













目錄

用Python做科学计算

用Python做科学计算

使用说明

本书使用reStructuredText编写，采用Sphinx发布。在此基础上添加了评论功能，你可以在hyry.dip.jp的在线版本中点击章节标题前面的评论按钮，对每个章节进行评论。推荐使用IE7.0以上、FireFox、Google Chome等浏览器阅读本书。

本书有两个镜像地址：

- http://hyry.dip.jp/pydoc (每日更新)
- http://pyscin.appspot.com/html/index.html (每周更新)

请使用下面的链接下载各种打包版本，其中Html打包版本格式最为正确，CHM和PDF版都多少有些问题。

下载Html打包版 下载CHM版 下载PDF版 下载源代码

另外，你还可以通过Google文档和 ZoomQuiet.org(国内下载快速)下载PDF版本

请查看 最近更新 了解最新添加的内容

关于HYRY Studio

- HYRY Studio首页： http://hyry.dip.jp
- 博客地址： http://hyry.dip.jp/blogt.py

Python是一种面向对象的、动态的程序设计语言。具有非常简洁而清晰的语法，适合于完成各种高层任务。它既可以用来快速开发程序脚本，也可以用来开发大规模的软件。

随着NumPy, SciPy, Matplotlib, Enthought librarys等众多程序库的开发，Python越来越适合于做科学计算、绘制高质量的2D和3D图像。和科学计算领域最流行的商业软件Matlab相比，Python是一门通用的程序设计语言，比Matlab所采用的脚本语言的应用范围更广泛，有更多的程序库的支持。虽然Matlab中的许多高级功能和toolbox目前还是无法替代的，不过在日常的科研开发之中仍然有很多的工作是可以用Python代劳的。

本书将介绍如何用Python开发科学计算的应用程序，除了介绍数值计算之外，我们还将着重介绍如何制作交互式的2D、3D图像；如何设计精巧的程序界面；如何和C语言所编写的高速计算程序结合；如何编写声音、图像处理算法。

阅读本书你需要学习过Python语言的一些基础知识，对面向对象的程序开发有所了解。有关Python语言基础的知识，可以参考：

啄木鸟社区的Python图书概览： http://wiki.woodpecker.org.cn/moin/PyBooks

本书中的所有示例均在Windows XP系统下采用Python(x,y)通过测试。如果你觉得安装众多的Python程序库很麻烦，不妨下载安装Python(x,y)。请阅读：软件包的安装和介绍

基础篇

科学计算所用到的各种库的入门介绍

手册篇

各个库的用户使用手册的翻译

实战篇

用所学到的东西解决实际问题

附录

源程序集

软件包的安装和介绍

安装软件包

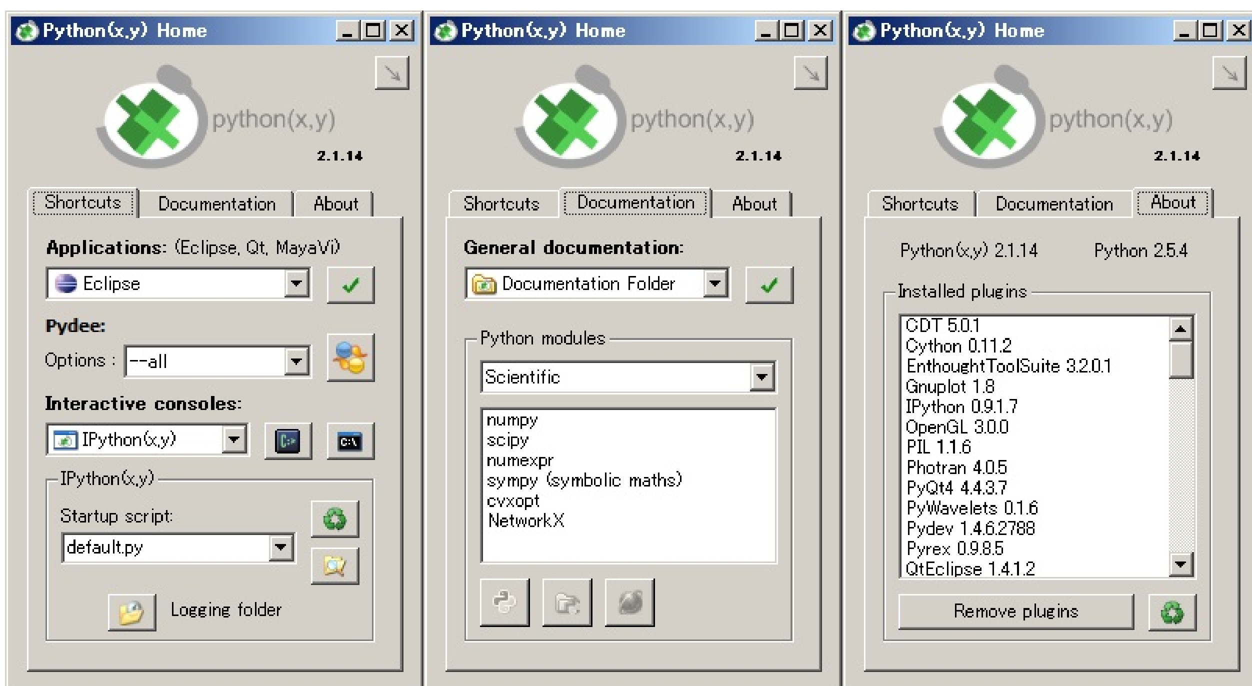
安装

和Matlab不同，Python的科学软件包由众多的社区维护和发布，因此要一一将其收集齐安装到你的电脑里是一件很费时间的事情。幸好这些工作已经有人帮我们整理好了。只需要下载一个文件，一次安装就能拥有众多的函数库可供使用。

这里介绍两个科学计算Python合集的下载和安装过程。

Python(x,y)

http://www.pythonxy.com 发布的Python(x,y)将近400M，收集了众多的函数库以及文档、教程。并且提供了一个方便的启动界面：



Python(x,y)的启动画面

- Shortcuts：启动各种应用程序
- Documentation：打开各个软件包的文档
- About：查看所安装的程序库的版本信息

Enthought Python Distribution (EPD)

下载地址：http://www.enthought.com/products/getepd.php EPD是一个商业的Python发行版本，同样包括了众多的科学软件包，而且作为教学使用是免费的，大小约为250M。

工具

安装好了之后先看看下面这些常用的工具，在以后的学习过程中会经常用到。

iPython

ipython 是一个 python 的交互式 shell，比默认的 python shell 好用得多，支持变量自动补全，自动缩近，支持 bash shell 命令，内置了许多很有用的功能和函数。

如果你安装了Python(x,y)的话，可以从Python(x,y)的启动界面中运行iPython。



通过Python(x,y) Home启动IPython的各种选项

从下拉选择框中选择你想运行的iPython，然后点击后面的①或者②按钮启动iPython。下拉选择框中的IPython(x,y)、IPython(Qt)、IPython(wxPython)和IPython(mlab)等几个选项都是启动iPython，只不过它们的启动方式不同。而Python选项则只启动单纯的Python Shell。

选项	参数	含义
IPython(x,y)	-pylab -p xy	
IPython(Qt)	-q4thread	
IPython(wxPython)	-wthread	
IPython(mlab)	-wthread	

点击①按钮将用一个叫做Console的软件启动Shell，此软件在窗口中显示Shell，并且支持多标签。点击②按钮用Windows自带的Cmd启动Shell。

如果你用python(x,y)的启动界面通过IPython(x,y)运行iPython的话，那么在iPython打开之后自动运行一个default.py脚本。此脚本缺省执行以下的函数库导入:

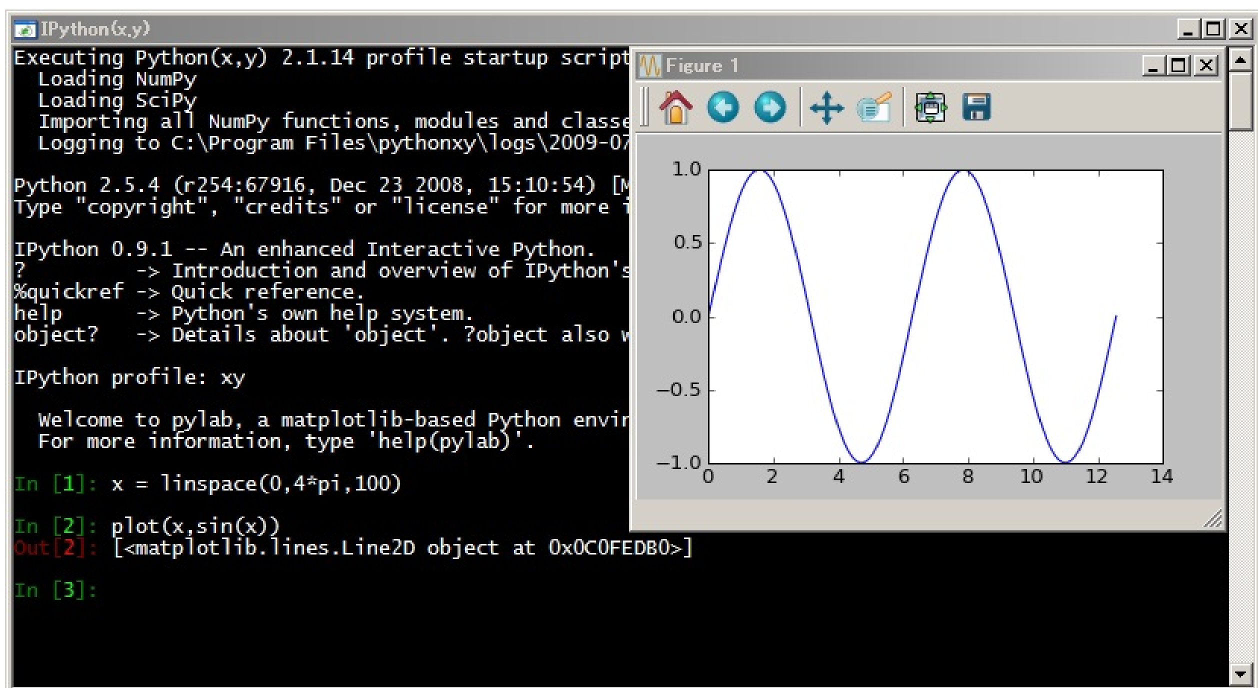
```
import numpy
import scipy
from numpy import *
```

为了和numpy, scipy等社区的推荐的标准导入方式一致，请点击按钮③，然后在打开的文件夹中添加一个名为numpy.py的文件，编辑此文件，添加以下几行推荐的导入:

```
import numpy as np
import scipy as sp
import pylab as pl
```

此后运行IPython(x,y)的时候请记着要选择numpy.py为启动脚本。

如果要使用pylab，TraitsUI等在shell中和图形界面进行交互的话，需要选择带-wthread参数的选项(-pylab也可以)。下图是一个用pylab绘制sin波形图的例子：



使用IPython交互式地绘制正弦波

在iPython的交互中可以方便地使用如下功能：

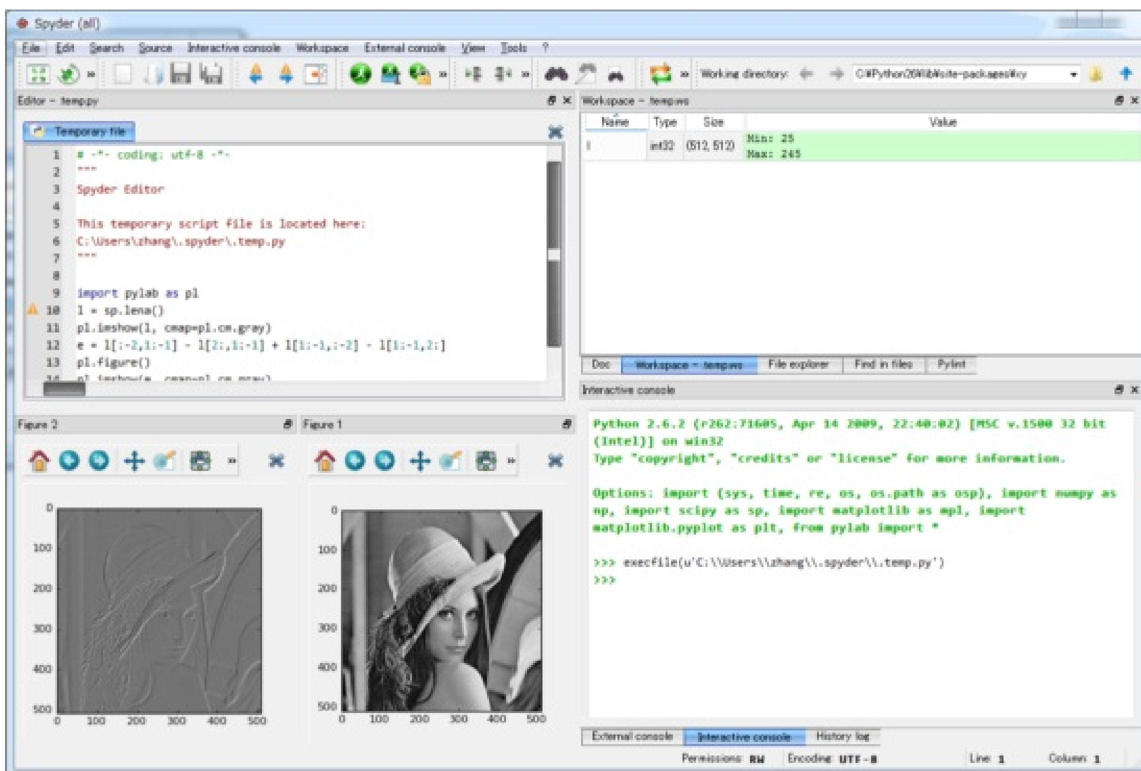
- 自动补全：输入一部分文字之后按tab键，iPython将列出所有以输入补全信息。
- 查看文档：输入需要查看文档的函数，然后在后面添加?或者??，?表示查看函数的文档，??表示查看其Python源代码，如果函数不是Python写的，则查看不到。
- 执行cmd命令：ls-列出当前目录下的所有文件，cd-显示或者更改当前路径
- 执行Python程序：用run *.py命令，在IPython中运行指定的py文件。如果加-i参数的话，则在IPython的命名空间中执行。也就是说在文件中没有定义名称会直接使用在IPython中的。
- 执行剪切板中的程序：你可以从本书中复制代码，然后在IPython命令窗口中执行paste命令运行复制的代码。如果执行paste foo的话，将把剪切板中的内容复制到变量foo中。变量foo是一个IPython提供的SList列表类型，它提供了很多操作所复制的内容的方法。
- 执行系统命令：在要执行的系统名字之前添加一个!号。例如执行!test.py的话，那么将让系统运行test.py文件。和run命令不同的是，test.py完全在另外的进程中运行。

spyder

spyder是Python(x,y)的作者为它开发的一个简单的Python开发环境。和其它Python IDE相比它最大的优点就是模仿MATLAB的workspace功能，可以很方便地观察和修改数组的值。

spyder的项目地址： http://code.google.com/p/spyderlib

下图是spyder的界面截图：

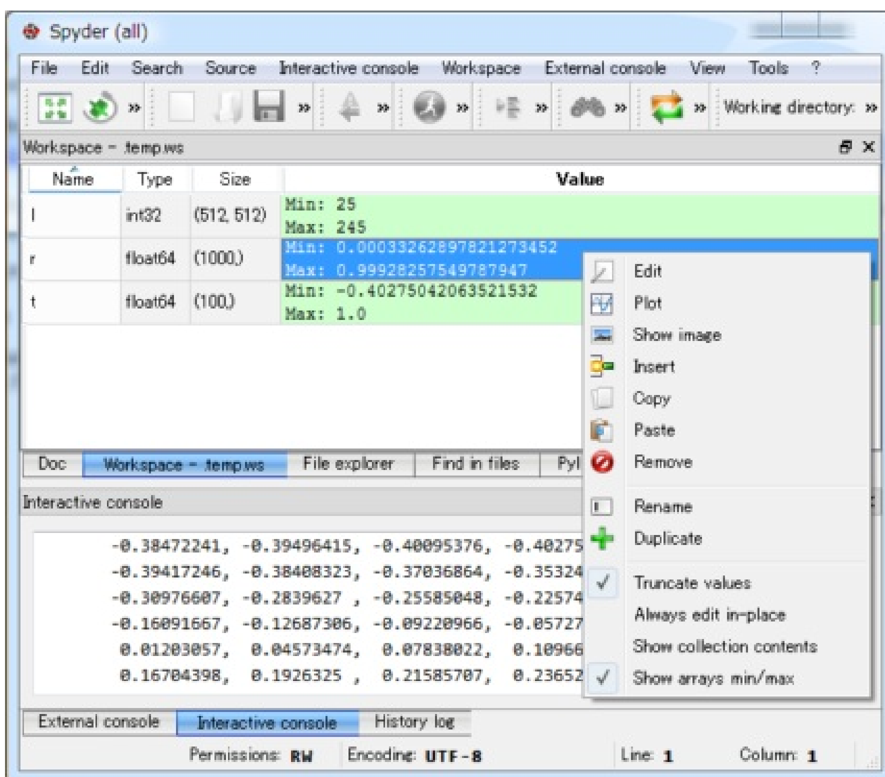


在Spyder中执行图像处理的程序

下图是Workspace的截图，列出了其中的变量名以及类型和大小等信息。鼠标右键可以显示出操作指定变量的菜单：

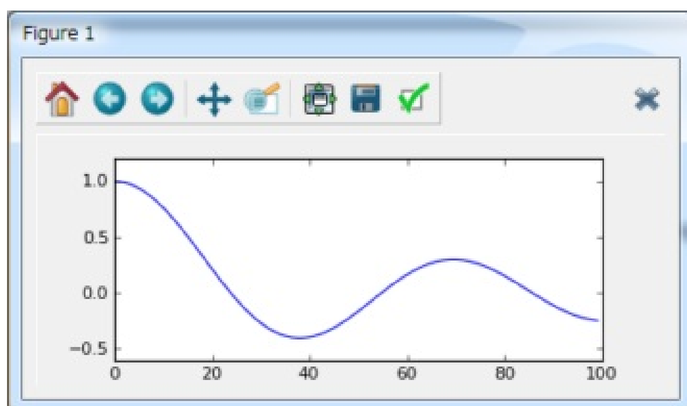
Warning

Workspace缺省配置不显示大写字母开头的变量，可以在Workspace菜单中修改这项配置。



使用Workspace查看变量内容

选择Plot选项，将出现如下图所示的绘图窗口：



在Workspace中将数组绘制成曲线图

如果Edit选项的话，将出现如下图所示的编辑器对数组进行操作：

Array editor - a

	0	1	2	3	4
0	0.000	1.000	2.000	3.000	4.000
1	10.000	11.000	12.000	13.000	14.000
2	20.000	21.000	22.000	23.000	24.000
3	30.000	31.000	32.000	33.000	34.000
4	40.000	41.000	42.000	43.000	44.000
5	50.000	51.000	52.000	53.000	54.000
6	60.000	61.000	62.000	63.000	64.000
7	70.000	71.000	72.000	73.000	74.000
8	80.000	81.000	82.000	83.000	84.000
9	90.000	91.000	92.000	93.000	94.000

Format Resize Background color

OK Cancel

使用数组编辑器查看和编辑数组内容

函数库介绍

Python的科学计算方面的内容由许多库构成，在基础篇中让我们首先来了解一下编写科学计算软件时经常使用的一些库。

数值计算库

NumPy为Python提供了快速的多维数组处理的能力，而**SciPy**则在NumPy基础上添加了众多的科学计算所需的各种工具包，有了这两个库，Python就有几乎和Matlab一样的处理数据和计算的能力了。

NumPy和SciPy官方网址： http://www.scipy.org

NumPy为Python带来了真正的多维数组功能，并且提供了丰富的函数库处理这些数组。它将常用的数学函数都进行数组化，使得这些数学函数能够直接对数组进行操作，将本来需要在Python级别进行的循环，放到C语言的运算中，明显地提高了程序的运算速度。

SciPy的核心计算部分都是一些久经考验的Fortran数值计算库，例如：

- 线性代数使用LAPACK库
- 快速傅立叶变换使用FFTPACK库
- 常微分方程求解使用ODEPACK库
- 非线性方程组求解以及最小值求解等使用MINPACK库

符号计算库

SymPy是一套进行符号数学运算的Python函数库，虽然它目前还没有到达1.0版本，但是已经足够好用，可以帮助我们进行公式推导，进行符号求解。

SymPy官方网址： http://code.google.com/p/sympy

界面设计

制作界面一直都是一件十分复杂的工作，使用**Traits**库，你将再也不会在界面设计上耗费大量精力，从而能把注意力集中到如何处理数据上去。

Traits官方网址： http://code.enthought.com/projects/traits

Traits库分为Traits和TraitsUI两大部分，Traits为Python添加了类型定义的功能，使用它定义的traits属性具有初始化、校验、代理、事件等诸多功能。

TraitsUI库基于Traits库，使用MVC结构快速地定义用户界面，在最简单的情况下，你甚至不需要写一句关于界面的代码，就可以通过traits属性定义获得一个可以工作的用户界面。使用TraitsUI库编写的程序自动支持wxPython和pyQt两个经典的界面库。

绘图与可视化

Chaco和**matplotlib**是很优秀的2D绘图库，Chaco库和Traits库紧密相连，方便制作动态交互式的图表功能。而matplotlib库则能够快速地绘制精美的图表、以多种格式输出，并且带有简单的3D绘图的功能。

Chaco官方网址： http://code.enthought.com/projects/chaco

matplotlib官方网址： http://matplotlib.sourceforge.net

TVTK库在标准的VTK库之上用Traits库进行封装，如果要在Python下使用VTK，用TVTK是再好不过的选择。**Mayavi2**则在TVTK的基础上再添加了一套面向应用的方便工具，它既可以单独作为3D可视化程序使用，也可以快速地嵌入到用户的程序中去。

Mayavi2官方网址： http://code.enthought.com/projects/mayavi

VTK(Visualization Toolkit)

视觉化工具函式库（VTK， Visualization Toolkit）是一个开放源码，跨平台、支援平行处理（VTK曾用于处理大小近乎1个Petabyte的资料，其平台为美国Los Alamos国家实验室所有的具1024个处理器之大型系统）的图形应用函式库。2005年实曾被美国陆军研究实验室用于即时模拟俄罗斯制反导弹战车ZSU23-4受到平面波攻击的情形，其计算节点高达2.5兆个之多。 -- 摘自维基百科

此外，使用**Visual**库能够快速、方便地制作3D动画演示，使你的数据结果更有说服力。

Visual官方网址： http://vpython.org

图像处理和计算机视觉

OpenCV是由英特尔公司发起并参与开发，以BSD许可证授权发行，可以在商业和研究领域中免费使用。OpenCV可用于开发实时的图像处理、计算机视觉以及模式识别程序。OpenCV提供的Python API方便我们快速实现算法，查看结果并且和其它的库进行数据交换。

NumPy-快速处理数据

标准安装的Python中用列表(list)保存一组值，可以用来当作数组使用，不过由于列表的元素可以是任何对象，因此列表中所保存的是对象的指针。这样为了保存一个简单的[1,2,3]，需要有3个指针和三个整数对象。对于数值运算来说这种结构显然比较浪费内存和CPU计算时间。

此外Python还提供了一个array模块，array对象和列表不同，它直接保存数值，和C语言的一维数组比较类似。但是由于它不支持多维，也没有各种运算函数，因此也不适合做数值运算。

NumPy的诞生弥补了这些不足，NumPy提供了两种基本的对象：ndarray（N-dimensional array object）和 ufunc（universal function object）。ndarray(下文统一称之为数组)是存储单一数据类型的多维数组，而ufunc则是能够对数组进行处理的函数。

ndarray对象

函数库的导入

本书的示例程序假设用以下推荐的方式导入NumPy函数库:

```
import numpy as np
```

创建

首先需要创建数组才能对其进行其它操作。

我们可以通过给array函数传递Python的序列对象创建数组，如果传递的是多层嵌套的序列，将创建多维数组(下例中的变量c):

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array((5, 6, 7, 8))
>>> c = np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]])
>>> b
array([5, 6, 7, 8])
>>> c
array([[1, 2, 3, 4],
 [4, 5, 6, 7],
 [7, 8, 9, 10]])
>>> c.dtype
dtype('int32')
```

数组的大小可以通过其shape属性获得：

```
>>> a.shape
(4,)
>>> c.shape
(3, 4)
```

数组a的shape只有一个元素，因此它是一维数组。而数组c的shape有两个元素，因此它是二维数组，其中第0轴的长度为3，第1轴的长度为4。还可以通过修改数组的shape属性，在保持数组元素个数不变的情况下，改变数组每个轴的长度。下面的例子将数组c的shape改为(4,3)，注意从(3,4)改为(4,3)并不是对数组进行转置，而只是改变每个轴的大小，数组元素在内存中的位置并没有改变：

```
>>> c.shape = 4,3
>>> c
array([[ 1,  2,  3],
 [ 4,  4,  5],
 [ 6,  7,  7],
 [ 8,  9, 10]])
```

当某个轴的元素为-1时，将根据数组元素的个数自动计算此轴的长度，因此下面的程序将数组c的shape改为了(2,6)：

```
>>> c.shape = 2,-1
>>> c
array([[ 1,  2,  3,  4,  4,  5],
 [ 6,  7,  7,  8,  9, 10]])
```

使用数组的reshape方法，可以创建一个改变了尺寸的新数组，原数组的shape保持不变：

```
>>> d = a.reshape((2,2))
>>> d
array([[1, 2],
 [3, 4]])
>>> a
array([1, 2, 3, 4])
```

数组a和d其实共享数据存储内存区域，因此修改其中任意一个数组的元素都会同时修改另外一个数组的内容：

```
>>> a[1] = 100 # 将数组a的第一个元素改为100
>>> d # 注意数组d中的2也被改变了
array([[  1, 100],
 [  3,   4]])
```

数组的元素类型可以通过dtype属性获得。上面例子中的参数序列的元素都是整数，因此所创建的数组的元素类型也是整数，并且是32bit的长整型。可以通过dtype参数在创建时指定元素类型:

```
>>> np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]], dtype=np.f
array([[  1.,   2.,   3.,   4.],
 [  4.,   5.,   6.,   7.],
 [  7.,   8.,   9.,  10.]])
>>> np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]], dtype=np.c
array([[  1.+0.j,   2.+0.j,   3.+0.j,   4.+0.j],
 [  4.+0.j,   5.+0.j,   6.+0.j,   7.+0.j],
 [  7.+0.j,   8.+0.j,   9.+0.j,  10.+0.j]])
```

上面的例子都是先创建一个Python序列，然后通过array函数将其转换为数组，这样做显然效率不高。因此NumPy提供了很多专门用来创建数组的函数。下面的每个函数都有一些关键字参数，具体用法请查看函数说明。

- arange函数类似于python的range函数，通过指定开始值、终值和步长来创建一维数组，注意数组不包括终值:

```
>>> np.arange(0,1,0.1)
array([ 0\. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,
```

- linspace函数通过指定开始值、终值和元素个数来创建一维数组，可以通过endpoint关键字指定是否包括终值，缺省设置是包括终值:

```
>>> np.linspace(0, 1, 12)
array([ 0\.        ,  0.09090909,  0.18181818,  0.27272727,  0.
 0.45454545,  0.54545455,  0.63636364,  0.72727273,  0.81818182
 0.90909091,  1\.        ])
```

- logspace函数和linspace类似，不过它创建等比数列，下面的例子产生1(10^0)到100(10^2)、有20个元素的等比数列:

```
>>> np.logspace(0, 2, 20)
array([   1\.        ,    1.27427499,    1.62377674,    2.06913
 2.6366509 ,    3.35981829,    4.2813324 ,    5.45559478,
 6.95192796,    8.8586679 ,   11.28837892,   14.38449888,
 18.32980711,   23.35721469,   29.76351442,   37.92690191,
 48.32930239,   61.58482111,   78.47599704,  100\.        ])
```

此外，使用frombuffer, fromstring, fromfile等函数可以从字节序列创建数组，下面以fromstring为例:

```
>>> s = "abcdefgh"
```

Python的字符串实际上是字节序列，每个字符占一个字节，因此如果从字符串s创建一个8bit的整数数组的话，所得到的数组正好就是字符串中每个字符的ASCII编码:

```
>>> np.fromstring(s, dtype=np.int8)
array([ 97,  98,  99, 100, 101, 102, 103, 104], dtype=int8)
```

如果从字符串s创建16bit的整数数组，那么两个相邻的字节就表示一个整数，把字节98和字节97当作一个16位的整数，它的值就是98*256+97 = 25185。可以看出内存中是以little endian(低位字节在前)方式保存数据的。

```
>>> np.fromstring(s, dtype=np.int16)
array([25185, 25699, 26213, 26727], dtype=int16)
>>> 98*256+97
25185
```

如果把整个字符串转换为一个64位的双精度浮点数数组，那么它的值是:

```
>>> np.fromstring(s, dtype=np.float)
array([  8.54088322e+194])
```

显然这个例子没有什么意义，但是可以想象如果我们用C语言的二进制方式写了一组double类型的数值到某个文件中，那们可以从此文件读取相应的数据，并通过fromstring函数将其转换为float64类型的数组。

我们可以写一个Python的函数，它将数组下标转换为数组中对应的值，然后使用此函数创建数组：

```
>>> def func(i):
...   return i%4+1
...
>>> np.fromfunction(func, (10,))
array([ 1.,  2.,  3.,  4.,  1.,  2.,  3.,  4.,  1.,  2.])
```

fromfunction函数的第一个参数为计算每个数组元素的函数，第二个参数为数组的大小(shape)，因为它支持多维数组，所以第二个参数必须是一个序列，本例中用(10,)创建一个10元素的一维数组。

下面的例子创建一个二维数组表示九九乘法表，输出的数组a中的每个元素a[i, j]都等于func2(i, j)：

```
>>> def func2(i, j):
...     return (i+1) * (j+1)
...
>>> a = np.fromfunction(func2, (9,9))
>>> a
array([[  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.],
 [  2.,   4.,   6.,   8.,  10.,  12.,  14.,  16.,  18.],
 [  3.,   6.,   9.,  12.,  15.,  18.,  21.,  24.,  27.],
 [  4.,   8.,  12.,  16.,  20.,  24.,  28.,  32.,  36.],
 [  5.,  10.,  15.,  20.,  25.,  30.,  35.,  40.,  45.],
 [  6.,  12.,  18.,  24.,  30.,  36.,  42.,  48.,  54.],
 [  7.,  14.,  21.,  28.,  35.,  42.,  49.,  56.,  63.],
 [  8.,  16.,  24.,  32.,  40.,  48.,  56.,  64.,  72.],
 [  9.,  18.,  27.,  36.,  45.,  54.,  63.,  72.,  81.]])
```

存取元素

数组元素的存取方法和Python的标准方法相同：

```
>>> a = np.arange(10)
>>> a[5]    # 用整数作为下标可以获取数组中的某个元素
5
>>> a[3:5]  # 用范围作为下标获取数组的一个切片，包括a[3]不包括a[5]
array([3, 4])
>>> a[:5]   # 省略开始下标，表示从a[0]开始
array([0, 1, 2, 3, 4])
>>> a[:-1]  # 下标可以使用负数，表示从数组后往前数
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> a[2:4] = 100,101    # 下标还可以用来修改元素的值
>>> a
array([  0,   1, 100, 101,   4,   5,   6,   7,   8,   9])
>>> a[1:-1:2]   # 范围中的第三个参数表示步长，2表示隔一个元素取一个元素
array([  1, 101,   5,   7])
>>> a[::-1] # 省略范围的开始下标和结束下标，步长为-1，整个数组头尾颠倒
array([  9,   8,   7,   6,   5,   4, 101, 100,   1,   0])
>>> a[5:1:-2] # 步长为负数时，开始下标必须大于结束下标
array([  5, 101])
```

和Python的列表序列不同，通过下标范围获取的新的数组是原始数组的一个视图。它与原始数组共享同一块数据空间：

```
>>> b = a[3:7] # 通过下标范围产生一个新的数组b，b和a共享同一块数据空间
>>> b
array([101,   4,   5,   6])
>>> b[2] = -10 # 将b的第2个元素修改为-10
>>> b
array([101,   4, -10,   6])
>>> a # a的第5个元素也被修改为10
array([  0,   1, 100, 101,   4, -10,   6,   7,   8,   9])
```

除了使用下标范围存取元素之外，NumPy还提供了两种存取元素的高级方法。

使用整数序列

当使用整数序列对数组元素进行存取时，将使用整数序列中的每个元素作为下标，整数序列可以是列表或者数组。使用整数序列作为下标获得的数组不和原始数组共享数据空间。

```
>>> x = np.arange(10,1,-1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[[3, 3, 1, 8]] # 获取x中的下标为3, 3, 1, 8的4个元素，组成一个新的数组
array([7, 7, 9, 2])
>>> b = x[np.array([3,3,-3,8])]  #下标可以是负数
>>> b[2] = 100
>>> b
array([7, 7, 100, 2])
>>> x   # 由于b和x不共享数据空间，因此x中的值并没有改变
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[[3,5,1]] = -1, -2, -3 # 整数序列下标也可以用来修改元素的值
>>> x
array([10, -3,  8, -1,  6, -2,  4,  3,  2])
```

使用布尔数组

当使用布尔数组b作为下标存取数组x中的元素时，将收集数组x中所有在数组b中对应下标为True的元素。使用布尔数组作为下标获得的数组不和原始数组共享数据空间，注意这种方式只对应于布尔数组，不能使用布尔列表。

```
>>> x = np.arange(5,0,-1)
>>> x
array([5, 4, 3, 2, 1])
>>> x[np.array([True, False, True, False, False])]
>>> # 布尔数组中下标为0，2的元素为True，因此获取x中下标为0,2的元素
array([5, 3])
>>> x[[True, False, True, False, False]]
>>> # 如果是布尔列表，则把True当作1, False当作0，按照整数序列方式获取x中的元
array([4, 5, 4, 5, 5])
>>> x[np.array([True, False, True, True])]
>>> # 布尔数组的长度不够时，不够的部分都当作False
array([5, 3, 2])
>>> x[np.array([True, False, True, True])] = -1, -2, -3
>>> # 布尔数组下标也可以用来修改元素
>>> x
array([-1,  4, -2, -3,  1])
```

布尔数组一般不是手工产生，而是使用布尔运算的ufunc函数产生，关于ufunc函数请参照 ufunc运算 一节。

```
>>> x = np.random.rand(10) # 产生一个长度为10，元素值为0-1的随机数的数组
>>> x
array([ 0.72223939,  0.921226  ,  0.7770805 ,  0.2055047 ,  0.1756
 0.95799412,  0.12015178,  0.7627083 ,  0.43260184,  0.91379859])
>>> x>0.5
>>> # 数组x中的每个元素和0.5进行大小比较，得到一个布尔数组，True表示x中对应的
array([ True,  True,  True, False, False,  True, False,  True, Fals
>>> x[x>0.5]
>>> # 使用x>0.5返回的布尔数组收集x中的元素，因此得到的结果是x中所有大于0.5的
array([ 0.72223939,  0.921226  ,  0.7770805 ,  0.95799412,  0.7627
 0.91379859])
```

多维数组

多维数组的存取和一维数组类似，因为多维数组有多个轴，因此它的下标需要用多个值来表示，NumPy采用组元(tuple)作为数组的下标。如下图所示，a为一个6x6的数组，图中用颜色区分了各个下标以及其对应的选择区域。

组元不需要圆括号

虽然我们经常在Python中用圆括号将组元括起来，但是其实组元的语法定义只需要用逗号隔开即可，例如 x,y=y,x 就是用组元交换变量值的一个例子。

```
>>> a[0,3:5]
array([3,4])
>>> a[4:,4:]
array([[44,45],[54,55]])
>>> a[:,2]
array([2,12,22,32,42,52])
>>> a[2::2,::2]
array([[20,22,24],
       [40,42,44]])
```

使用数组切片语法访问多维数组中的元素

如何创建这个数组

你也许会对如何创建a这样的数组感到好奇，数组a实际上是一个加法表，纵轴的值为0, 10, 20, 30, 40, 50；横轴的值为0, 1, 2, 3, 4, 5。纵轴的每个元素都和横轴的每个元素求和，就得到图中所示的数组a。你可以用下面的语句创建它，至于其原理我们将在后面的章节进行讨论：

```
>>> np.arange(0, 60, 10).reshape(-1, 1) + np.arange(0, 6)
array([[ 0,  1,  2,  3,  4,  5],
 [10, 11, 12, 13, 14, 15],
 [20, 21, 22, 23, 24, 25],
 [30, 31, 32, 33, 34, 35],
 [40, 41, 42, 43, 44, 45],
 [50, 51, 52, 53, 54, 55]])
```

多维数组同样也可以使用整数序列和布尔数组进行存取。

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([1,12,23,34,45])
>>> a[3:,[0,2,5]]
array([[30,32,35],
       [40,42,45],
       [50,52,55]])
>>> mask=np.array([1,0,1,0,0,1],
          dtype=np.bool)
>>> a[mask,2]
array([2,22,52])
```

第0轴

第 1 轴

使用整数序列和布尔数组访问多维数组中的元素

- a[(0,1,2,3,4),(1,2,3,4,5)]：用于存取数组的下标和仍然是一个有两个元素的组元，组元中的每个元素都是整数序列，分别对应数组的第0轴和第1轴。从两个序列的对应位置取出两个整数组成下标： a[0,1], a[1,2], ..., a[4,5]。

- a[3:, [0, 2, 5]] : 下标中的第0轴是一个范围，它选取第3行之后的所有行；第1轴是整数序列，它选取第0, 2, 5三列。
- a[mask, 2] : 下标的第0轴是一个布尔数组，它选取第0，2，5行；第1轴是一个整数，选取第2列。

结构数组

在C语言中我们可以通过struct关键字定义结构类型，结构中的字段占据连续的内存空间，每个结构体占用的内存大小都相同，因此可以很容易地定义结构数组。和C语言一样，在NumPy中也很容易对这种结构数组进行操作。只要NumPy中的结构定义和C语言中的定义相同，NumPy就可以很方便地读取C语言的结构数组的二进制数据，转换为NumPy的结构数组。

假设我们需要定义一个结构数组，它的每个元素都有name, age和weight字段。在NumPy中可以如下定义：

```
import numpy as np
persontype = np.dtype({
    'names':['name', 'age', 'weight'],
    'formats':['S32','i', 'f']})
a = np.array([("Zhang",32,75.5),("Wang",24,65.2)],
    dtype=persontype)
```

我们先创建一个dtype对象persontype，通过其字典参数描述结构类型的各个字段。字典有两个关键字：names，formats。每个关键字对应的值都是一个列表。names定义结构中的每个字段名，而formats则定义每个字段的类型：

- **S32** : 32个字节的字符串类型，由于结构中的每个元素的大小必须固定，因此需要指定字符串的长度
- **i** : 32bit的整数类型，相当于np.int32
- **f** : 32bit的单精度浮点数类型，相当于np.float32

然后我们调用array函数创建数组，通过关键字参数 dtype=persontype， 指定所创建的数组的元素类型为结构persontype。运行上面程序之后，我们可以在IPython中执行如下的语句查看数组a的元素类型

```
>>> a.dtype
dtype([('name', '|S32'), ('age', '<i4'), ('weight', '<f4')])
```

这里我们看到了另外一种描述结构类型的方法： 一个包含多个组元的列表，其中形如 (字段名, 类型描述) 的组元描述了结构中的每个字段。类型描述前面为我们添加了 '|', '<' 等字符，这些字符用来描述字段值的字节顺序：

- **|** : 忽视字节顺序
- **<** : 低位字节在前
- **>** : 高位字节在前

结构数组的存取方式和一般数组相同，通过下标能够取得其中的元素，注意元素的值看上去像是组元，实际上它是一个结构：

```
>>> a[0]
('Zhang', 32, 75.5)
>>> a[0].dtype
dtype([('name', '|S32'), ('age', '<i4'), ('weight', '<f4')])
```

a[0]是一个结构元素，它和数组a共享内存数据，因此可以通过修改它的字段，改变原始数组中的对应字段：

```
>>> c = a[1]
>>> c["name"] = "Li"
>>> a[1]["name"]
"Li"
```

结构像字典一样可以通过字符串下标获取其对应的字段值：

```
>>> a[0]["name"]
'Zhang'
```

我们不但可以获得结构元素的某个字段，还可以直接获得结构数组的字段，它返回的是原始数组的视图，因此可以通过修改b[0]改变a[0]["age"]：

```
>>> b=a[:]["age"] # 或者a["age"]
>>> b
array([32, 24])
>>> b[0] = 40
>>> a[0]["age"]
40
```

通过调用a.tostring或者a.tofile方法，可以直接输出数组a的二进制形式：

```
>>> a.tofile("test.bin")
```

利用下面的C语言程序可以将test.bin文件中的数据读取出来。

内存对齐

C语言的结构体为了内存寻址方便，会自动的添加一些填充用的字节，这叫做内存对齐。例如如果把下面的name[32]改为name[30]的话，由于内存对齐问题，在name和age中间会填补两个字节，最终的结构体大小不会改变。因此如果numpy中

的所配置的内存大小不符合C语言的对齐规范的话，将会出现数据错位。为了解决这个问题，在创建dtype对象时，可以传递参数align=True，这样numpy的结构数组的内存对齐和C语言的结构体就一致了。

```
#include <stdio.h>

struct person
{
    char name[32];
    int age;
    float weight;
};

struct person p[2];

void main ()
{
    FILE *fp;
    int i;
    fp=fopen("test.bin","rb");
    fread(p, sizeof(struct person), 2, fp);
    fclose(fp);
    for(i=0;i<2;i++)
        printf("%s %d %f\n", p[i].name, p[i].age, p[i].weight);
    getchar();
}
```

结构类型中可以包括其它的结构类型，下面的语句创建一个有一个字段f1的结构，f1的值是另外一个结构，它有字段f2，其类型为16bit整数。

```
>>> np.dtype([('f1', [('f2', np.int16)])])
dtype([('f1', [('f2', '<i2')])])
```

当某个字段类型为数组时，用组元的第三个参数表示，下面描述的f1字段是一个shape为(2,3)的双精度浮点数组:

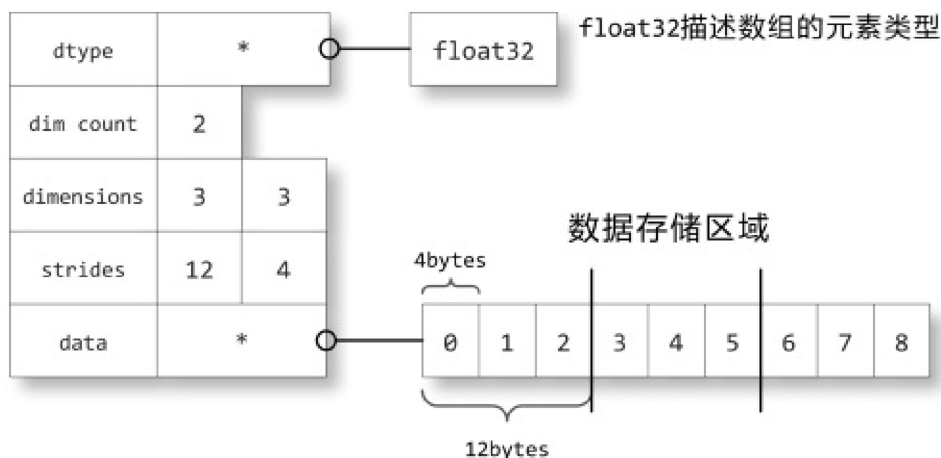
```
>>> np.dtype([('f0', 'i4'), ('f1', 'f8', (2, 3))])
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
```

用下面的字典参数也可以定义结构类型，字典的关键字为结构中字段名，值为字段的类型描述，但是由于字典的关键字是没有顺序的，因此字段的顺序需要在类型描述中给出，类型描述是一个组元，它的第二个值给出字段的字节为单位的偏移量，例如age字段的偏移量为25个字节：

```
>>> np.dtype({'surname':('S25',0),'age':(np.uint8,25)})
dtype([('surname', '|S25'), ('age', '|u1')])
```

内存结构

下面让我们来看看ndarray数组对象是如何在内存中储存的。如下图所示，关于数组的描述信息保存在一个数据结构中，这个结构引用两个对象：一块用于保存数据的存储区域和一个用于描述元素类型的dtype对象。



ndarray数组对象在内存中的储存方式

数据存储区域保存着数组中所有元素的二进制数据，dtype对象则知道如何将元素的二进制数据转换为可用的值。数组的维数、大小等信息都保存在ndarray数组对象的数据结构中。图中显示的是如下数组的内存结构：

```
>>> a = np.array([[0,1,2],[3,4,5],[6,7,8]], dtype=np.float32)
```

strides中保存的是当每个轴的下标增加1时，数据存储区中的指针所增加的字节数。例如图中的strides为12,4，即第0轴的下标增加1时，数据的地址增加12个字节：即a[1,0]的地址比a[0,0]的地址要高12个字节，正好是3个单精度浮点数的总字节数；第1轴下标增加1时，数据的地址增加4个字节，正好是单精度浮点数的字节数。

如果strides中的数值正好和对应轴所占据的字节数相同的话，那么数据在内存中是连续存储的。然而数据并不一定都是连续储存的，前面介绍过通过下标范围得到新的数组是原始数组的视图，即它和原始视图共享数据存储区域：

```
>>> b = a[::2,::2]
>>> b
array([[ 0.,  2.],
 [ 6.,  8.]], dtype=float32)
>>> b.strides
(24, 8)
```

由于数组b和数组a共享数据存储区，而b中的第0轴和第1轴都是数组a中隔一个元素取一个，因此数组b的strides变成了24,8，正好都是数组a的两倍。 对照前面的图很容易看出数据0和2的地址相差8个字节，而0和6的地址相差24个字节。

元素在数据存储区中的排列格式有两种：C语言格式和Fortan语言格式。在C语言中，多维数组的第0轴是最上位的，即第0轴的下标增加1时，元素的地址增加的字节数最多；而Fortan语言的多维数组的第0轴是最下位的，即第0轴的下标增加1时，地址只增加一个元素的字节数。在NumPy中，元素在内存中的排列缺省是以C语言格式存储的，如果你希望改为Fortan格式的话，只需要给数组传递order="F"参数：

```
>>> c = np.array([[0,1,2],[3,4,5],[6,7,8]], dtype=np.float32, order
>>> c.strides
(4, 12)
```

ufunc运算

ufunc是universal function的缩写，它是一种能对数组的每个元素进行操作的函数。NumPy内置的许多ufunc函数都是在C语言级别实现的，因此它们的计算速度非常快。让我们来看一个例子：

```
>>> x = np.linspace(0, 2*np.pi, 10)
# 对数组x中的每个元素进行正弦计算，返回一个同样大小的新数组
>>> y = np.sin(x)
>>> y
array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
  8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
 -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
 -2.44921271e-16])
```

先用linspace产生一个从0到2*PI的等距离的10个数，然后将其传递给sin函数，由于np.sin是一个ufunc函数，因此它对x中的每个元素求正弦值，然后将结果返回，并且赋值给y。计算之后x中的值并没有改变，而是新创建了一个数组保存结果。如果我们希望将sin函数所计算的结果直接覆盖到数组x上去的话，可以将要被覆盖的数组作为第二个参数传递给ufunc函数。例如：:

```
>>> t = np.sin(x,x)
>>> x
array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
  8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
 -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
 -2.44921271e-16])
>>> id(t) == id(x)
True
```

sin函数的第二个参数也是x，那么它所做的事情就是对x中的每给值求正弦值，并且把结果保存到x中的对应的位置中。此时函数的返回值仍然是整个计算的结果，只不过它就是x，因此两个变量的id是相同的(变量t和变量x指向同一块内存区域)。

我用下面这个小程序，比较了一下numpy.math和Python标准库的math.sin的计算速度：:

```
import time
import math
import numpy as np

x = [i * 0.001 for i in xrange(1000000)]
start = time.clock()
for i, t in enumerate(x):
    x[i] = math.sin(t)
print "math.sin:", time.clock() - start

x = [i * 0.001 for i in xrange(1000000)]
x = np.array(x)
start = time.clock()
np.sin(x,x)
print "numpy.sin:", time.clock() - start

# 输出
# math.sin: 1.15426932753
# numpy.sin: 0.0882399858083
```

在我的电脑上计算100万次正弦值，numpy.sin比math.sin快10倍多。这得利于numpy.sin在C语言级别的循环计算。numpy.sin同样也支持对单个数值求正弦，例如：numpy.sin(0.5)。不过值得注意的是，对单个数的计算math.sin则比numpy.sin快得多了，让我们看下面这个测试程序：

```
x = [i * 0.001 for i in xrange(1000000)]
start = time.clock()
for i, t in enumerate(x):
    x[i] = np.sin(t)
print "numpy.sin loop:", time.clock() - start

# 输出
# numpy.sin loop: 5.72166965355
```

请注意numpy.sin的计算速度只有math.sin的1/5。这是因为numpy.sin为了同时支持数组和单个值的计算，其C语言的内部实现要比math.sin复杂很多，如果我们同样在Python级别进行循环的话，就会看出其中的差别了。此外，numpy.sin返回的数的类型和math.sin返回的类型有所不同，math.sin返回的是Python的标准float类型，而numpy.sin则返回一个numpy.float64类型：

```
>>> type(math.sin(0.5))
<type 'float'>
>>> type(np.sin(0.5))
<type 'numpy.float64'>
```

通过上面的例子我们了解了如何最有效率地使用math库和numpy库中的数学函数。因为它们各有长短，因此在导入时不建议使用*号全部载入，而是应该使用import numpy as np的方式载入，这样我们可以根据需要选择合适的函数调用。

NumPy中有众多的ufunc函数为我们提供各式各样的计算。除了sin这种单输入函数之外，还有许多多个输入的函数，add函数就是一个最常用的例子。先来看一个例子:

```
>>> a = np.arange(0,4)
>>> a
array([0, 1, 2, 3])
>>> b = np.arange(1,5)
>>> b
array([1, 2, 3, 4])
>>> np.add(a,b)
array([1, 3, 5, 7])
>>> np.add(a,b,a)
array([1, 3, 5, 7])
>>> a
array([1, 3, 5, 7])
```

add函数返回一个新的数组，此数组的每个元素都为两个参数数组的对应元素之和。它接受第3个参数指定计算结果所要写入的数组，如果指定的话，add函数就不再产生新的数组。

由于Python的操作符重载功能，计算两个数组相加可以简单地写为a+b，而np.add(a,b,a)则可以用a+=b来表示。下面是数组的运算符和其对应的ufunc函数的一个列表，注意除号"/"的意义根据是否激活**future**.division有所不同。

y = x1 + x2:	add(x1, x2 [, y])
y = x1 - x2:	subtract(x1, x2 [, y])
y = x1 * x2:	multiply (x1, x2 [, y])
y = x1 / x2:	divide (x1, x2 [, y]), 如果两个数组的元素为整数，那么用整数除法
y = x1 / x2:	true divide (x1, x2 [, y]), 总是返回精确的商
y = x1 // x2:	floor divide (x1, x2 [, y]), 总是对返回值取整
y = -x:	negative(x [,y])
y = x1**x2:	power(x1, x2 [, y])
y = x1 % x2:	remainder(x1, x2 [, y]), mod(x1, x2, [, y])

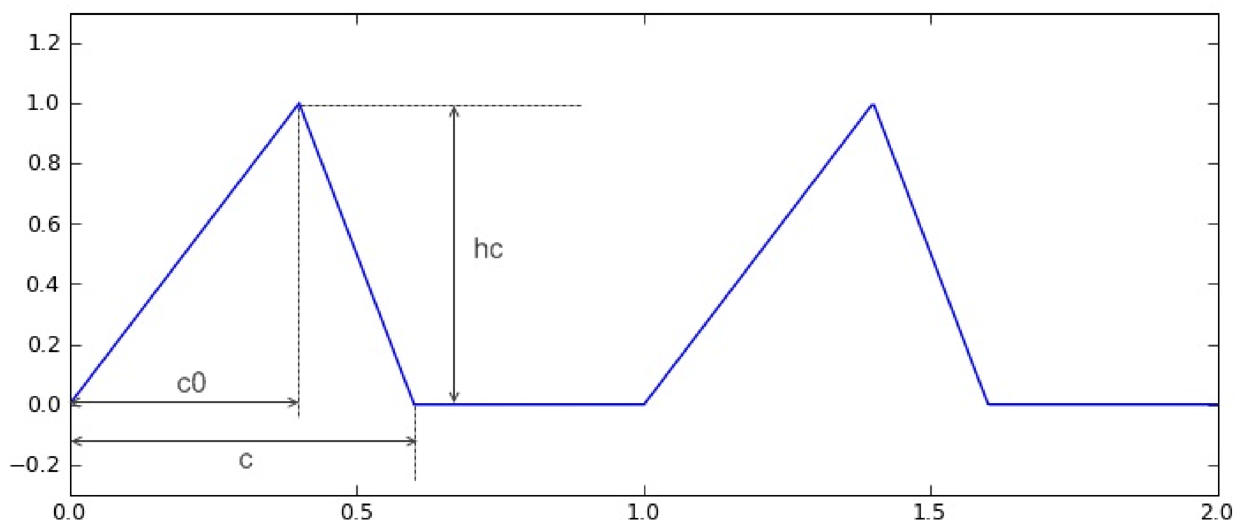
数组对象支持这些操作符，极大地简化了算式的编写，不过要注意如果你的算式很复杂，并且要运算的数组很大的话，会因为产生大量的中间结果而降低程序的运算效率。例如：假设a b c三个数组采用算式x=a*b+c计算，那么它相当于:

```
t = a * b
x = t + c
del t
```

也就是说需要产生一个数组t保存乘法的计算结果，然后再产生最后的结果数组x。我们可以通过手工将一个算式分解为x = a*b; x += c，以减少一次内存分配。

通过组合标准的ufunc函数的调用，可以实现各种算式的数组计算。不过有些时候这种算式不易编写，而针对每个元素的计算函数却很容易用Python实现，这时可以用frompyfunc函数将一个计算单个元素的函数转换成ufunc函数。这样就可以方便地用所产生的ufunc函数对数组进行计算了。让我们来看一个例子。

我们想用一个分段函数描述三角波，三角波的样子如下图所示：



三角波可以用分段函数进行计算

我们很容易根据上图所示写出如下的计算三角波某点y坐标的函数:

```
def triangle_wave(x, c, c0, hc):
    x = x - int(x) # 三角波的周期为1，因此只取x坐标的小数部分进行计算
    if x >= c: r = 0.0
    elif x < c0: r = x / c0 * hc
    else: r = (c-x) / (c-c0) * hc
    return r
```

显然triangle_wave函数只能计算单个数值，不能对数组直接进行处理。我们可以用下面的方法先使用列表包容(List comprehension)，计算出一个list，然后用array函数将列表转换为数组：

```
x = np.linspace(0, 2, 1000)
y = np.array([triangle_wave(t, 0.6, 0.4, 1.0) for t in x])
```

这种做法每次都需要使用列表包容语法调用函数，对于多维数组是很麻烦的。让我们来看看如何用frompyfunc函数来解决这个问题：

```
triangle_ufunc = np.frompyfunc( lambda x: triangle_wave(x, 0.6, 0.4
y2 = triangle_ufunc(x)
```

frompyfunc的调用格式为frompyfunc(func, nin, nout)，其中func是计算单个元素的函数，nin是此函数的输入参数的个数，nout是此函数的返回值的个数。虽然triangle_wave函数有4个参数，但是由于后三个c, c0, hc在整个计算中值都是固定的，因此所产生的ufunc函数其实只有一个参数。为了满足这个条件，我们用一个lambda函数对triangle_wave的参数进行一次包装。这样传入frompyfunc的函数就只有一个参数了。这样子做，效率并不是太高，另外还有一种方法：

```
def triangle_func(c, c0, hc):
    def trifunc(x):
        x = x - int(x) # 三角波的周期为1，因此只取x坐标的小数部分进行计算
        if x >= c: r = 0.0
        elif x < c0: r = x / c0 * hc
        else: r = (c-x) / (c-c0) * hc
        return r

    # 用trifunc函数创建一个ufunc函数，可以直接对数组进行计算, 不过通过此函数
    # 计算得到的是一个Object数组，需要进行类型转换
    return np.frompyfunc(trifunc, 1, 1)

y2 = triangle_func(0.6, 0.4, 1.0)(x)
```

我们通过函数triangle_func包装三角波的三个参数，在其内部定义一个计算三角波的函数trifunc，trifunc函数在调用时会采用triangle_func的参数进行计算。最后triangle_func返回用frompyfunc转换结果。

值得注意的是用frompyfunc得到的函数计算出的数组元素的类型为object，因为frompyfunc函数无法保证Python函数返回的数据类型都完全一致。因此还需要再次y2.astype(np.float64)将其转换为双精度浮点数组。

广播

当我们使用ufunc函数对两个数组进行计算时，ufunc函数会对这两个数组的对应元素进行计算，因此它要求这两个数组有相同的大小(shape相同)。如果两个数组的shape不同的话，会进行如下的广播(broadcasting)处理：

1. 让所有输入数组都向其中shape最长的数组看齐，shape中不足的部分都通过在前面加1补齐
2. 输出数组的shape是输入数组shape的各个轴上的最大值
3. 如果输入数组的某个轴和输出数组的对应轴的长度相同或者其长度为1时，这个数组能够用来计算，否则出错
4. 当输入数组的某个轴的长度为1时，沿着此轴运算时都用此轴上的第一组值

上述4条规则理解起来可能比较费劲，让我们来看一个实际的例子。

先创建一个二维数组a，其shape为(6,1)：

```
>>> a = np.arange(0, 60, 10).reshape(-1, 1)
>>> a
array([[ 0], [10], [20], [30], [40], [50]])
>>> a.shape
(6, 1)
```

再创建一维数组b，其shape为(5,)：

```
>>> b = np.arange(0, 5)
>>> b
array([0, 1, 2, 3, 4])
>>> b.shape
(5,)
```

计算a和b的和，得到一个加法表，它相当于计算a,b中所有元素组的和，得到一个shape为(6,5)的数组：

```
>>> c = a + b
>>> c
array([[ 0,  1,  2,  3,  4],
 [10, 11, 12, 13, 14],
 [20, 21, 22, 23, 24],
 [30, 31, 32, 33, 34],
 [40, 41, 42, 43, 44],
 [50, 51, 52, 53, 54]])
>>> c.shape
(6, 5)
```

由于a和b的shape长度(也就是ndim属性)不同，根据规则1，需要让b的shape向a对齐，于是将b的shape前面加1，补齐为(1,5)。相当于做了如下计算：

```
>>> b.shape=1,5
>>> b
array([[0, 1, 2, 3, 4]])
```

这样加法运算的两个输入数组的shape分别为(6,1)和(1,5)，根据规则2，输出数组的各个轴的长度为输入数组各个轴上的长度的最大值，可知输出数组的shape为(6,5)。

由于b的第0轴上的长度为1，而a的第0轴上的长度为6，因此为了让它们在第0轴上能够相加，需要将b在第0轴上的长度扩展为6，这相当于：

```
>>> b = b.repeat(6,axis=0)
>>> b
array([[0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4]])
```

由于a的第1轴的长度为1，而b的第一轴长度为5，因此为了让它们在第1轴上能够相加，需要将a在第1轴上的长度扩展为5，这相当于：

```
>>> a = a.repeat(5, axis=1)
>>> a
array([[ 0,  0,  0,  0,  0],
 [10, 10, 10, 10, 10],
 [20, 20, 20, 20, 20],
 [30, 30, 30, 30, 30],
 [40, 40, 40, 40, 40],
 [50, 50, 50, 50, 50]])
```

经过上述处理之后，a和b就可以按对应元素进行相加运算了。

当然，numpy在执行a+b运算时，其内部并不会真正将长度为1的轴用repeat函数进行扩展，如果这样做的话就太浪费空间了。

由于这种广播计算很常用，因此numpy提供了一个快速产生如上面a,b数组的方法：ogrid对象：

```
>>> x,y = np.ogrid[0:5,0:5]
>>> x
array([[0],
 [1],
 [2],
 [3],
 [4]])
>>> y
array([[0, 1, 2, 3, 4]])
```

ogrid是一个很有趣的对象，它像一个多维数组一样，用切片组元作为下标进行存取，返回的是一组可以用来广播计算的数组。其切片下标有两种形式：

- 开始值:结束值:步长，和np.arange(开始值, 结束值, 步长)类似
- 开始值:结束值:长度j，当第三个参数为虚数时，它表示返回的数组的长度，和np.linspace(开始值, 结束值, 长度)类似：

  ```
  >>> x, y = np.ogrid[0:1:4j, 0:1:3j]
  >>> x
  array([[ 0\.        ],
   [ 0.33333333],
   [ 0.66666667],
   [ 1\.        ]])
  >>> y
  array([[ 0\. ,  0.5,  1\. ]])
  ```

ogrid为什么不是函数

根据Python的语法，只有在中括号中才能使用用冒号隔开的切片语法，如果ogrid是函数的话，那么这些切片必须使用slice函数创建，这显然会增加代码的长度。

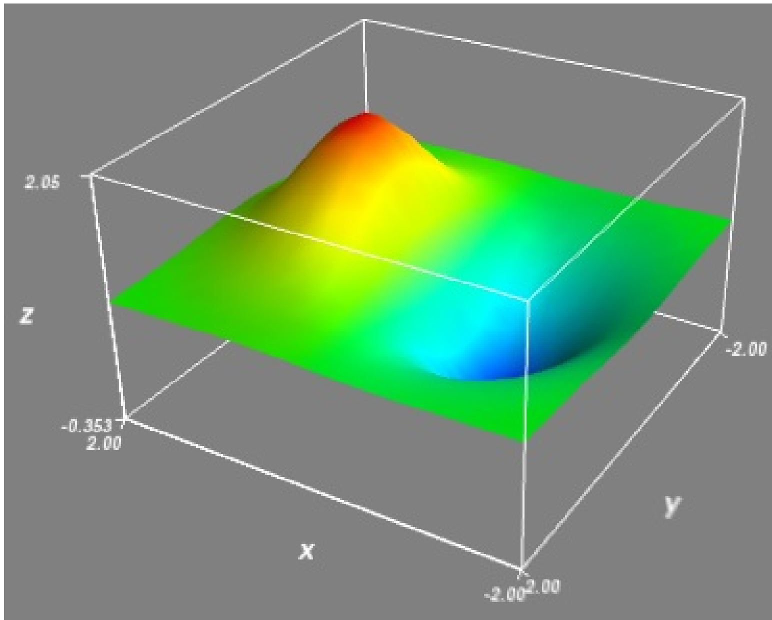
利用ogrid的返回值，我能很容易计算x, y网格面上各点的值，或者x, y, z网格体上各点的值。下面是绘制三维曲面 x exp(x*2 - y2) 的程序：

```
import numpy as np
from enthought.mayavi import mlab

x, y = np.ogrid[-2:2:20j, -2:2:20j]
z = x * np.exp( - x**2 - y**2)

pl = mlab.surf(x, y, z, warp_scale="auto")
mlab.axes(xlabel='x', ylabel='y', zlabel='z')
mlab.outline(pl)
```

此程序使用mayavi的mlab库快速绘制如下图所示的3D曲面，关于mlab的相关内容将在今后的章节进行介绍。



使用ogrid创建的三维曲面

ufunc的方法

ufunc函数本身还有些方法，这些方法只对两个输入一个输出的ufunc函数有效，其它的ufunc对象调用这些方法时会抛出ValueError异常。

reduce 方法和Python的reduce函数类似，它沿着axis轴对array进行操作，相当于将<op>运算符插入到沿axis轴的所有子数组或者元素当中。

```
>>> np.add.reduce([1,2,3]) # 1 + 2 + 3
6
>>> np.add.reduce([[1,2,3],[4,5,6]], axis=1) # 1,4 + 2,5 + 3,6
array([ 6, 15])
```

accumulate 方法和reduce方法类似，只是它返回的数组和输入的数组的shape相同，保存所有的中间计算结果：

```
>>> np.add.accumulate([1,2,3])
array([1, 3, 6])
>>> np.add.accumulate([[1,2,3],[4,5,6]], axis=1)
array([[ 1,  3,  6],
 [ 4,  9, 15]])
```

reduceat 方法计算多组reduce的结果，通过indices参数指定一系列reduce的起始和终了位置。reduceat的计算有些特别，让我们通过一个例子来解释一下：

```
>>> a = np.array([1,2,3,4])
>>> result = np.add.reduceat(a,indices=[0,1,0,2,0,3,0])
>>> result
array([ 1,  2,  3,  3,  6,  4, 10])
```

对于indices中的每个元素都会调用reduce函数计算出一个值来，因此最终计算结果的长度和indices的长度相同。结果result数组中除最后一个元素之外，都按照如下计算得出：

```
np.reduce(a[indices[-1]:])
```

因此上面例子中，结果的每个元素如下计算而得：

```
>>> a.shape += (1,)*b.ndim
>>> <op>(a,b)
>>> a = a.squeeze()
```

其中squeeze的功能是剔除数组a中长度为1的轴。如果你看不太明白这个等同程序的话，让我们来看一个例子：

```
>>> np.multiply.outer([1,2,3,4,5],[2,3,4])
array([[ 2,  3,  4],
 [ 4,  6,  8],
 [ 6,  9, 12],
 [ 8, 12, 16],
 [10, 15, 20]])
```

可以看出通过outer方法计算的结果是如下的乘法表：

```
#    2, 3, 4
# 1
# 2
# 3
# 4
# 5
```

如果将这两个数组按照等同程序一步一步的计算的话，就会发现乘法表最终是通过广播的方式计算出来的。

矩阵运算

NumPy和Matlab不一样，对于多维数组的运算，缺省情况下并不使用矩阵运算，如果你希望对数组进行矩阵运算的话，可以调用相应的函数。

matrix对象

numpy库提供了matrix类，使用matrix类创建的是矩阵对象，它们的加减乘除运算缺省采用矩阵方式计算，因此用法和matlab十分类似。但是由于NumPy中同时存在ndarray和matrix对象，因此用户很容易将两者弄混。这有违Python的“显式优于隐式”的原则，因此并不推荐在较复杂的程序中使用matrix。下面是使用matrix的一个例子：

```
>>> a = np.matrix([[1,2,3],[5,5,6],[7,9,9]])
>>> a*a**-1
matrix([[  1.00000000e+00,   1.66533454e-16,  -8.32667268e-17],
 [ -2.77555756e-16,   1.00000000e+00,  -2.77555756e-17],
 [  1.66533454e-16,   5.55111512e-17,   1.00000000e+00]])
```

因为a是用matrix创建的矩阵对象，因此乘法和幂运算符都变成了矩阵运算，于是上面计算的是矩阵a和其逆矩阵的乘积，结果是一个单位矩阵。

矩阵的乘积可以使用dot函数进行计算。对于二维数组，它计算的是矩阵乘积，对于一维数组，它计算的是其点积。当需要将一维数组当作列矢量或者行矢量进行矩阵运算时，推荐先使用reshape函数将一维数组转换为二维数组：

```
>>> a = array([1, 2, 3])
>>> a.reshape((-1,1))
array([[1],
 [2],
 [3]])
>>> a.reshape((1,-1))
array([[1, 2, 3]])
```

除了dot计算乘积之外，NumPy还提供了inner和outer等多种计算乘积的函数。这些函数计算乘积的方式不同，尤其是当对于多维数组的时候，更容易搞混。

- **dot** : 对于两个一维的数组，计算的是这两个数组对应下标元素的乘积和(数学上称之为内积)；对于二维数组，计算的是两个数组的矩阵乘积；对于多维数组，它的通用计算公式如下，即结果数组中的每个元素都是：数组a的最后一维上的所有元素与数组b的倒数第二位上的所有元素的乘积和：

> ```
> dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
> ```
>
> 下面以两个3为数组的乘积演示一下dot乘积的计算结果：
>
> 首先创建两个3维数组，这两个数组的最后两维满足矩阵乘积的条件：
>
> ```
> >>> a = np.arange(12).reshape(2,3,2)
> >>> b = np.arange(12,24).reshape(2,2,3)
> >>> c = np.dot(a,b)
> ```
>
> dot乘积的结果c可以看作是数组a,b的多个子矩阵的乘积：
>
> ```
> >>> np.alltrue( c[0,:,0,:] == np.dot(a[0],b[0]) )
> True
> >>> np.alltrue( c[1,:,0,:] == np.dot(a[1],b[0]) )
> True
> >>> np.alltrue( c[0,:,1,:] == np.dot(a[0],b[1]) )
> True
> >>> np.alltrue( c[1,:,1,:] == np.dot(a[1],b[1]) )
> True
> ```

- **inner** : 和dot乘积一样，对于两个一维数组，计算的是这两个数组对应下标元素的乘积和；对于多维数组，它计算的结果数组中的每个元素都是：数组a和b的最后一维的内积，因此数组a和b的最后一维的长度必须相同：

> ```
> inner(a, b)[i,j,k,m] = sum(a[i,j,:]*b[k,m,:])
> ```
>
> 下面是inner乘积的演示：
>
> ```
> >>> a = np.arange(12).reshape(2,3,2)
> >>> b = np.arange(12,24).reshape(2,3,2)
> >>> c = np.inner(a,b)
> >>> c.shape
> (2, 3, 2, 3)
> >>> c[0,0,0,0] == np.inner(a[0,0],b[0,0])
> True
> >>> c[0,1,1,0] == np.inner(a[0,1],b[1,0])
> True
> >>> c[1,2,1,2] == np.inner(a[1,2],b[1,2])
> True
> ```

- **outer** : 只按照一维数组进行计算，如果传入参数是多维数组，则先将此数组展平为一维数组之后再进行运算。outer乘积计算的列向量和行向量的矩阵乘积：

> ```
> >>> np.outer([1,2,3],[4,5,6,7])
> array([[ 4,  5,  6,  7],
>  [ 8, 10, 12, 14],
>  [12, 15, 18, 21]])
> ```

矩阵中更高级的一些运算可以在NumPy的线性代数子库linalg中找到。例如inv函数计算逆矩阵，solve函数可以求解多元一次方程组。下面是solve函数的一个例子：

```
>>> a = np.random.rand(10,10)
>>> b = np.random.rand(10)
>>> x = np.linalg.solve(a,b)
>>> np.sum(np.abs(np.dot(a,x) - b))
3.1433189384699745e-15
```

solve函数有两个参数a和b。a是一个N*N的二维数组，而b是一个长度为N的一维数组，solve函数找到一个长度为N的一维数组x，使得a和x的矩阵乘积正好等于b，数组x就是多元一次方程组的解。

有关线性代数方面的内容将在今后的章节中详细介绍。

文件存取

NumPy提供了多种文件操作函数方便我们存取数组内容。文件存取的格式分为两类：二进制和文本。而二进制格式的文件又分为NumPy专用的格式化二进制类型和无格式类型。

使用数组的方法函数tofile可以方便地将数组中数据以二进制的格式写进文件。tofile输出的数据没有格式，因此用numpy.fromfile读回来的时候需要自己格式化数据：

```
>>> a = np.arange(0,12)
>>> a.shape = 3,4
>>> a
array([[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]])
>>> a.tofile("a.bin")
>>> b = np.fromfile("a.bin", dtype=np.float) # 按照float类型读入数据
>>> b # 读入的数据是错误的
array([  2.12199579e-314,   6.36598737e-314,   1.06099790e-313,
 1.48539705e-313,   1.90979621e-313,   2.33419537e-313])
>>> a.dtype # 查看a的dtype
dtype('int32')
>>> b = np.fromfile("a.bin", dtype=np.int32) # 按照int32类型读入数据
>>> b # 数据是一维的
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> b.shape = 3, 4 # 按照a的shape修改b的shape
>>> b # 这次终于正确了
array([[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]])
```

从上面的例子可以看出，需要在读入的时候设置正确的dtype和shape才能保证数据一致。并且tofile函数不管数组的排列顺序是C语言格式的还是Fortran语言格式的，统一使用C语言格式输出。

此外如果fromfile和tofile函数调用时指定了sep关键字参数的话，数组将以文本格式输入输出。

numpy.load和numpy.save函数以NumPy专用的二进制类型保存数据，这两个函数会自动处理元素类型和shape等信息，使用它们读写数组就方便多了，但是numpy.save输出的文件很难和其它语言编写的程序读入：

```
>>> np.save("a.npy", a)
>>> c = np.load( "a.npy" )
>>> c
array([[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]])
```

如果你想将多个数组保存到一个文件中的话，可以使用numpy.savez函数。savez函数的第一个参数是文件名，其后的参数都是需要保存的数组，也可以使用关键字参数为数组起一个名字，非关键字参数传递的数组会自动起名为arr_0, arr_1, ...。

savez函数输出的是一个压缩文件(扩展名为npz)，其中每个文件都是一个save函数保存的npy文件，文件名对应于数组名。load函数自动识别npz文件，并且返回一个类似于字典的对象，可以通过数组名作为关键字获取数组的内容：

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> b = np.arange(0, 1.0, 0.1)
>>> c = np.sin(b)
>>> np.savez("result.npz", a, b, sin_array = c)
>>> r = np.load("result.npz")
>>> r["arr_0"] # 数组a
array([[1, 2, 3],
 [4, 5, 6]])
>>> r["arr_1"] # 数组b
array([ 0\. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9]
>>> r["sin_array"] # 数组c
array([ 0\.        ,  0.09983342,  0.19866933,  0.29552021,  0.3894
 0.47942554,  0.56464247,  0.64421769,  0.71735609,  0.78332691])
```

如果你用解压软件打开result.npz文件的话，会发现其中有三个文件：arr_0.npy，arr_1.npy， sin_array.npy，其中分别保存着数组a, b, c的内容。

使用numpy.savetxt和numpy.loadtxt可以读写1维和2维的数组：

```
>>> a = np.arange(0,12,0.5).reshape(4,-1)
>>> np.savetxt("a.txt", a) # 缺省按照'%.18e'格式保存数据，以空格分隔
>>> np.loadtxt("a.txt")
array([[  0\. ,   0.5,   1\. ,   1.5,   2\. ,   2.5],
 [  3\. ,   3.5,   4\. ,   4.5,   5\. ,   5.5],
 [  6\. ,   6.5,   7\. ,   7.5,   8\. ,   8.5],
 [  9\. ,   9.5,  10\. ,  10.5,  11\. ,  11.5]])
>>> np.savetxt("a.txt", a, fmt="%d", delimiter=",") #改为保存为整数,
>>> np.loadtxt("a.txt",delimiter=",") # 读入的时候也需要指定逗号分隔
array([[  0.,   0.,   1.,   1.,   2.,   2.],
 [  3.,   3.,   4.,   4.,   5.,   5.],
 [  6.,   6.,   7.,   7.,   8.,   8.],
 [  9.,   9.,  10.,  10.,  11.,  11.]])
```

文件名和文件对象

本节介绍所举的例子都是传递的文件名，也可以传递已经打开的文件对象，例如对于load和save函数来说，如果使用文件对象的话，可以将多个数组储存到一个npy文件中：

```
>>> a = np.arange(8)
>>> b = np.add.accumulate(a)
>>> c = a + b
>>> f = file("result.npy", "wb")
>>> np.save(f, a) # 顺序将a,b,c保存进文件对象f
>>> np.save(f, b)
>>> np.save(f, c)
>>> f.close()
>>> f = file("result.npy", "rb")
>>> np.load(f) # 顺序从文件对象f中读取内容
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> np.load(f)
array([ 0,  1,  3,  6, 10, 15, 21, 28])
>>> np.load(f)
array([ 0,  2,  5,  9, 14, 20, 27, 35])
```

SciPy-数值计算库

SciPy函数库在NumPy库的基础上增加了众多的数学、科学以及工程计算中常用的库函数。例如线性代数、常微分方程数值求解、信号处理、图像处理、稀疏矩阵等等。由于其涉及的领域众多、本书没有能力对其一一的进行介绍。作为入门介绍，让我们看看如何用SciPy进行插值处理、信号滤波以及用C语言加速计算。

最小二乘拟合

假设有一组实验数据(x[i], y[i])，我们知道它们之间的函数关系:y = f(x)，通过这些已知信息，需要确定函数中的一些参数项。例如，如果f是一个线型函数f(x) = k*x*+*b*，那么参数*k*和*b*就是我们需要确定的值。如果将这些参数用 ***p*** 表示的话，那么我们就是要找到一组 **p* 值使得如下公式中的S函数最小：

$$S(\mathbf{p}) = \sum_{i=1}^{m} [y_i - f(x_i, \mathbf{p})]^2$$

这种算法被称之为最小二乘拟合(Least-square fitting)。

scipy中的子函数库optimize已经提供了实现最小二乘拟合算法的函数leastsq。下面是用leastsq进行数据拟合的一个例子：

```
# -*- coding: utf-8 -*-
import numpy as np
from scipy.optimize import leastsq
import pylab as pl

def func(x, p):
    """
    数据拟合所用的函数: A*sin(2*pi*k*x + theta)
    """
    A, k, theta = p
    return A*np.sin(2*np.pi*k*x+theta)

def residuals(p, y, x):
    """
    实验数据x, y和拟合函数之间的差，p为拟合需要找到的系数
    """
    return y - func(x, p)

x = np.linspace(0, -2*np.pi, 100)
A, k, theta = 10, 0.34, np.pi/6 # 真实数据的函数参数
y0 = func(x, [A, k, theta]) # 真实数据
y1 = y0 + 2 * np.random.randn(len(x)) # 加入噪声之后的实验数据

p0 = [7, 0.2, 0] # 第一次猜测的函数拟合参数

# 调用leastsq进行数据拟合
# residuals为计算误差的函数
# p0为拟合参数的初始值
# args为需要拟合的实验数据
plsq = leastsq(residuals, p0, args=(y1, x))

print u"真实参数:", [A, k, theta]
print u"拟合参数", plsq[0] # 实验数据拟合后的参数

pl.plot(x, y0, label=u"真实数据")
pl.plot(x, y1, label=u"带噪声的实验数据")
pl.plot(x, func(x, plsq[0]), label=u"拟合数据")
pl.legend()
pl.show()
```

这个例子中我们要拟合的函数是一个正弦波函数，它有三个参数 **A**, **k**, **theta**， 分别对应振幅、频率、相角。假设我们的实验数据是一组包含噪声的数据 x, y1，其中y1是在真实数据y0的基础上加入噪声的到了。

通过leastsq函数对带噪声的实验数据x, y1进行数据拟合，可以找到x和真实数据y0之间的正弦关系的三个参数： A, k, theta。下面是程序的输出：

```
# -*- coding: utf-8 -*-
# 本程序用各种fmin函数求卷积的逆运算

import scipy.optimize as opt
import numpy as np

def test_fmin_convolve(fminfunc, x, h, y, yn, x0):
    """
 x (*) h = y, (*)表示卷积
 yn为在y的基础上添加一些干扰噪声的结果
 x0为求解x的初始值
 """
    def convolve_func(h):
        """
 计算 yn - x (*) h 的power
 fmin将通过计算使得此power最小
 """
        return np.sum((yn - np.convolve(x, h))**2)

    # 调用fmin函数，以x0为初始值
    h0 = fminfunc(convolve_func, x0)

    print fminfunc.__name__
    print "---------------------"
    # 输出 x (*) h0 和 y 之间的相对误差
    print "error of y:", np.sum((np.convolve(x, h0)-y)**2)/np.sum(y
    # 输出 h0 和 h 之间的相对误差
    print "error of h:", np.sum((h0-h)**2)/np.sum(h**2)
    print

def test_n(m, n, nscale):
    """
 随机产生x, h, y, yn, x0等数列，调用各种fmin函数求解b
 m为x的长度, n为h的长度, nscale为干扰的强度
 """
    x = np.random.rand(m)
    h = np.random.rand(n)
    y = np.convolve(x, h)
    yn = y + np.random.rand(len(y)) * nscale
    x0 = np.random.rand(n)

    test_fmin_convolve(opt.fmin, x, h, y, yn, x0)
    test_fmin_convolve(opt.fmin_powell, x, h, y, yn, x0)
    test_fmin_convolve(opt.fmin_cg, x, h, y, yn, x0)
    test_fmin_convolve(opt.fmin_bfgs, x, h, y, yn, x0)

if __name__ == "__main__":
    test_n(200, 20, 0.1)
```

下面是程序的输出：

```
fmin
-------------
error of y: 0.00568756699607
error of h: 0.354083287918

fmin_powell
-------------
error of y: 0.000116114709857
error of h: 0.000258897894009

fmin_cg
-------------
error of y: 0.000111220299615
error of h: 0.000211404733439

fmin_bfgs
-------------
error of y: 0.000111220251551
error of h: 0.000211405138529
```

非线性方程组求解

optimize库中的fsolve函数可以用来对非线性方程组进行求解。它的基本调用形式如下：

```
fsolve(func, x0)
```

func(x)是计算方程组误差的函数，它的参数x是一个矢量，表示方程组的各个未知数的一组可能解，func返回将x代入方程组之后得到的误差；x0为未知数矢量的初始值。如果要对如下方程组进行求解的话：

- f1(u1,u2,u3) = 0
- f2(u1,u2,u3) = 0
- f3(u1,u2,u3) = 0

那么func可以如下定义：

```
def func(x):
    u1,u2,u3 = x
    return [f1(u1,u2,u3), f2(u1,u2,u3), f3(u1,u2,u3)]
```

下面是一个实际的例子，求解如下方程组的解：

- 5*x1 + 3 = 0
- 4*x0*x0 - 2*sin(x1*x2) = 0
- x1*x2 - 1.5 = 0

程序如下：

```
from scipy.optimize import fsolve
from math import sin,cos

def f(x):
    x0 = float(x[0])
    x1 = float(x[1])
    x2 = float(x[2])
    return [
        5*x1+3,
        4*x0*x0 - 2*sin(x1*x2),
        x1*x2 - 1.5
    ]

result = fsolve(f, [1,1,1])

print result
print f(result)
```

输出为：

```
[-0.70622057 -0.6         -2.5       ]
[0.0, -9.1260332624187868e-14, 5.3290705182007514e-15]
```

由于fsolve函数在调用函数f时，传递的参数为数组，因此如果直接使用数组中的元素计算的话，计算速度将会有所降低，因此这里先用float函数将数组中的元素转换为Python中的标准浮点数，然后调用标准math库中的函数进行运算。

在对方程组进行求解时，fsolve会自动计算方程组的雅可比矩阵，如果方程组中的未知数很多，而与每个方程有关的未知数较少时，即雅可比矩阵比较稀疏时，传递一个计算雅可比矩阵的函数将能大幅度提高运算速度。笔者在一个模拟计算的程序中需要大量求解近有50个未知数的非线性方程组的解。每个方程平均与6个未知数相关，通过传递雅可比矩阵的计算函数使计算速度提高了4倍。

雅可比矩阵

雅可比矩阵是一阶偏导数以一定方式排列的矩阵，它给出了可微分方程与给定点的最优线性逼近，因此类似于多元函数的导数。例如前面的函数f1,f2,f3和未知数u1,u2,u3的雅可比矩阵如下：

$$\begin{bmatrix} \frac{\partial f1}{\partial u1} & \frac{\partial f1}{\partial u2} & \frac{\partial f1}{\partial u3} \\ \frac{\partial f2}{\partial u1} & \frac{\partial f2}{\partial u2} & \frac{\partial f2}{\partial u3} \\ \frac{\partial f3}{\partial u1} & \frac{\partial f3}{\partial u2} & \frac{\partial f3}{\partial u3} \end{bmatrix}$$

使用雅可比矩阵的fsolve实例如下，计算雅可比矩阵的函数j通过fprime参数传递给fsolve，函数j和函数f一样，有一个未知数的解矢量参数x，函数j计算非线性方程组在矢量x点上的雅可比矩阵。由于这个例子中未知数很少，因此程序计算雅可比矩阵并不能带来计算速度的提升。

```
# -*- coding: utf-8 -*-
from scipy.optimize import fsolve
from math import sin,cos
def f(x):
    x0 = float(x[0])
    x1 = float(x[1])
    x2 = float(x[2])
    return [
        5*x1+3,
        4*x0*x0 - 2*sin(x1*x2),
        x1*x2 - 1.5
    ]

def j(x):
    x0 = float(x[0])
    x1 = float(x[1])
    x2 = float(x[2])
    return [
        [0, 5, 0],
        [8*x0, -2*x2*cos(x1*x2), -2*x1*cos(x1*x2)],
        [0, x2, x1]
    ]

result = fsolve(f, [1,1,1], fprime=j)
print result
print f(result)
```

B-Spline样条曲线

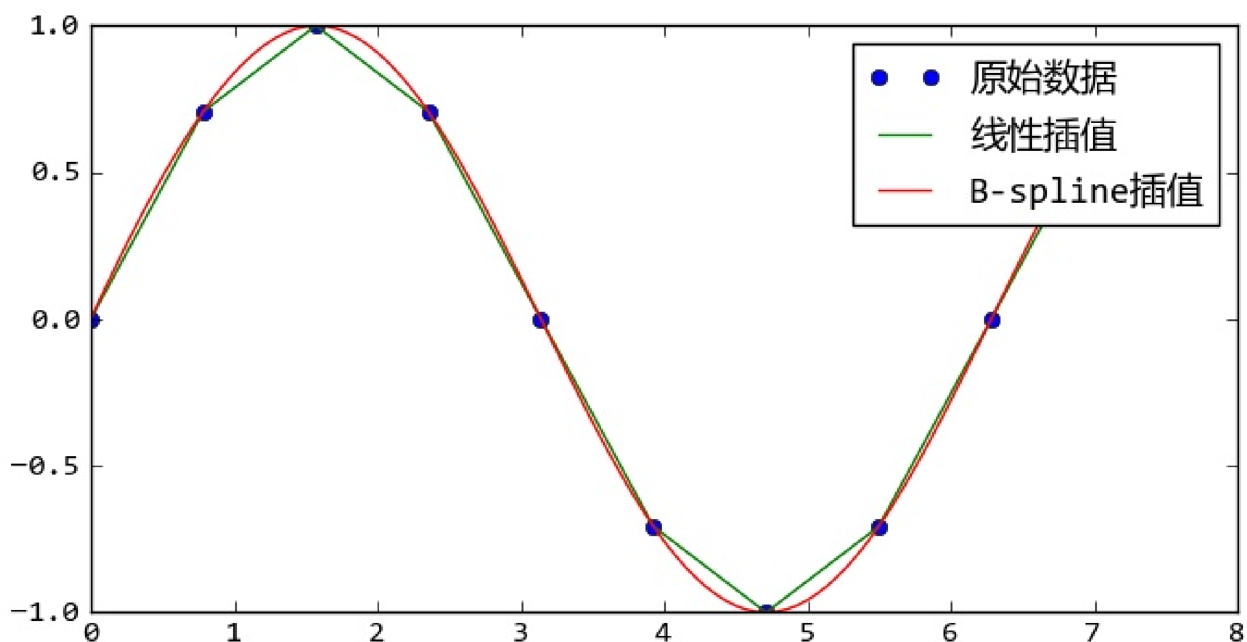
interpolate库提供了许多对数据进行插值运算的函数。下面是使用直线和B-Spline对正弦波上的点进行插值的例子。

```
# -*- coding: utf-8 -*-
import numpy as np
import pylab as pl
from scipy import interpolate

x = np.linspace(0, 2*np.pi+np.pi/4, 10)
y = np.sin(x)

x_new = np.linspace(0, 2*np.pi+np.pi/4, 100)
f_linear = interpolate.interp1d(x, y)
tck = interpolate.splrep(x, y)
y_bspline = interpolate.splev(x_new, tck)

pl.plot(x, y, "o",  label=u"原始数据")
pl.plot(x_new, f_linear(x_new), label=u"线性插值")
pl.plot(x_new, y_bspline, label=u"B-spline插值")
pl.legend()
pl.show()
```



使用interpolate库对正弦波数据进行线性插值和B-Spline插值

在这段程序中，通过interp1d函数直接得到一个新的线性插值函数。而B-Spline插值运算需要先使用splrep函数计算出B-Spline曲线的参数，然后将参数传递给splev函数计算出各个取样点的插值结果。

数值积分

数值积分是对定积分的数值求解，例如可以利用数值积分计算某个形状的面积。下面让我们来考虑一下如何计算半径为1的半圆的面积，根据圆的面积公式，其面积应该等于PI/2。单位半圆曲线可以用下面的函数表示：

```
def half_circle(x):
    return (1-x**2)**0.5
```

下面的程序使用经典的分小矩形计算面积总和的方式，计算出单位半圆的面积：

```
>>> N = 10000
>>> x = np.linspace(-1, 1, N)
>>> dx = 2.0/N
>>> y = half_circle(x)
>>> dx * np.sum(y[:-1] + y[1:]) # 面积的两倍
3.1412751679988937
```

利用上述方式计算出的圆上一系列点的坐标，还可以用numpy.trapz进行数值积分：

```
>>> import numpy as np
>>> np.trapz(y, x) * 2 # 面积的两倍
3.1415893269316042
```

此函数计算的是以x,y为顶点坐标的折线与X轴所夹的面积。同样的分割点数，trapz函数的结果更加接近精确值一些。

如果我们调用scipy.integrate库中的quad函数的话，将会得到非常精确的结果：

```
>>> from scipy import integrate
>>> pi_half, err = integrate.quad(half_circle, -1, 1)
>>> pi_half*2
3.1415926535897984
```

多重定积分的求值可以通过多次调用quad函数实现，为了调用方便，integrate库提供了dblquad函数进行二重定积分，tplquad函数进行三重定积分。下面以计算单位半球体积为例说明dblquad函数的用法。

单位半球上的点(x,y,z)符合如下方程：

$$x^2 + y^2 + z^2 = 1$$

因此可以如下定义通过(x,y)坐标计算球面上点的z值的函数：

```
def half_sphere(x, y):
    return (1-x**2-y**2)**0.5
```

X-Y轴平面与此球体的交线为一个单位圆，因此积分区间为此单位圆，可以考虑为X轴坐标从-1到1进行积分，而Y轴从 -half_circle(x) 到 half_circle(x) 进行积分，于是可以调用dblquad函数：

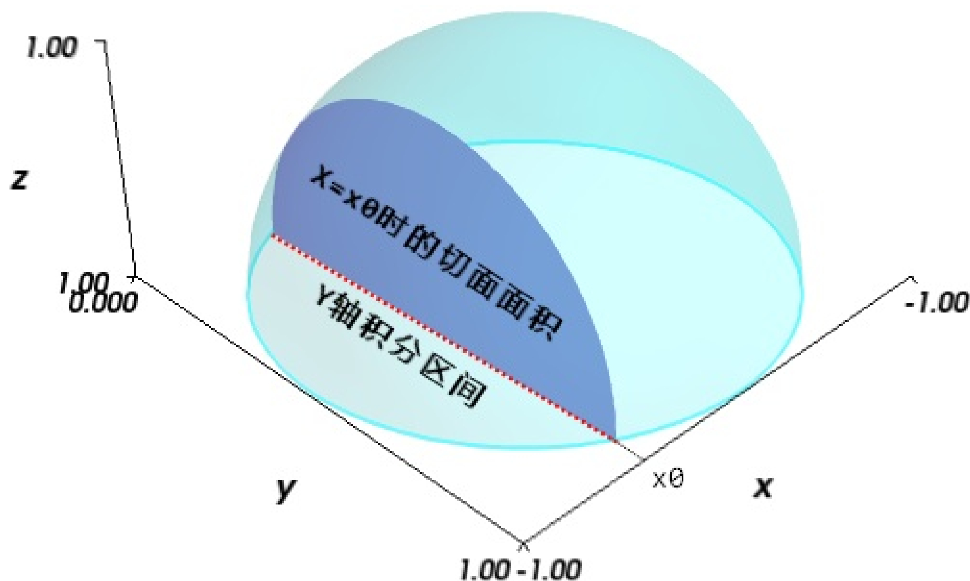
```
>>> integrate.dblquad(half_sphere, -1, 1,
 lambda x:-half_circle(x),
 lambda x:half_circle(x))
>>> (2.0943951023931988, 2.3252456653390915e-14)
>>> np.pi*4/3/2 # 通过球体体积公式计算的半球体积
2.0943951023931953
```

dblquad函数的调用方式为：

```
dblquad(func2d, a, b, gfun, hfun)
```

对于func2d(x,y)函数进行二重积分，其中a,b为变量x的积分区间，而gfun(x)到hfun(x)为变量y的积分区间。

半球体积的积分的示意图如下：



半球体积的双重定积分示意图

X轴的积分区间为-1.0到1.0，对于X=x0时，通过对Y轴的积分计算出切面的面积，因此Y轴的积分区间如图中红色点线所示。

解常微分方程组

scipy.integrate库提供了数值积分和常微分方程组求解算法odeint。下面让我们来看看如何用odeint计算洛仑兹吸引子的轨迹。洛仑兹吸引子由下面的三个微分方程定义：

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

洛仑兹吸引子的详细介绍: http://bzhang.lamost.org/website/archives/lorenz_attactor

这三个方程定义了三维空间中各个坐标点上的速度矢量。从某个坐标开始沿着速度矢量进行积分，就可以计算出无质量点在此空间中的运动轨迹。其中 σ, ρ, β 为三个常数，不同的参数可以计算出不同的运动轨迹： x(t), y(t), z(t)。 当参数为某些值时，轨迹出现馄饨现象：即微小的初值差别也会显著地影响运动轨迹。下面是洛仑兹吸引子的轨迹计算和绘制程序：

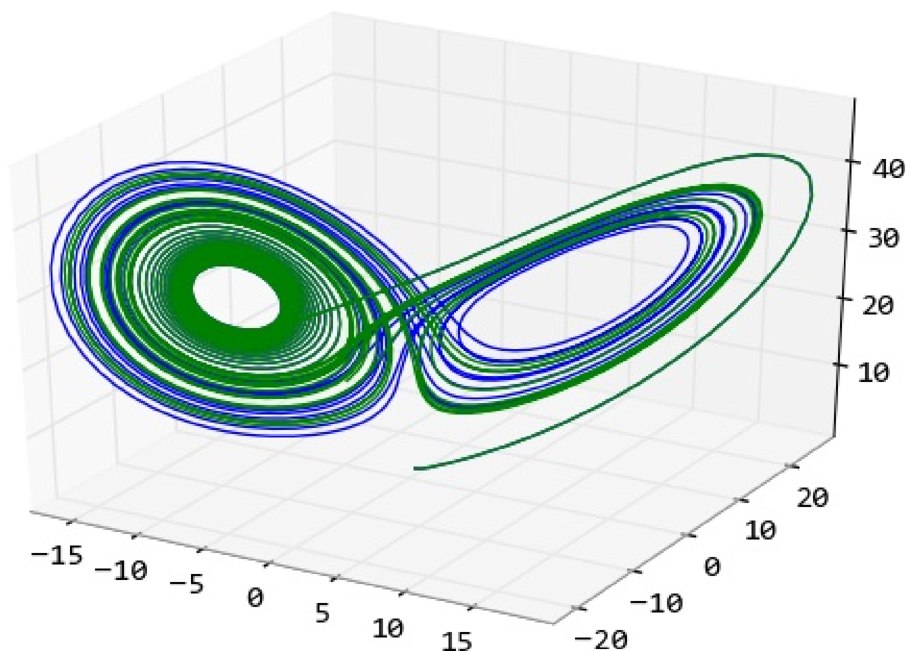
```
# -*- coding: utf-8 -*-
from scipy.integrate import odeint
import numpy as np

def lorenz(w, t, p, r, b):
    # 给出位置矢量w，和三个参数p, r, b计算出
    # dx/dt, dy/dt, dz/dt的值
    x, y, z = w
    # 直接与lorenz的计算公式对应
    return np.array([p*(y-x), x*(r-z)-y, x*y-b*z])

t = np.arange(0, 30, 0.01) # 创建时间点
# 调用ode对lorenz进行求解, 用两个不同的初始值
track1 = odeint(lorenz, (0.0, 1.00, 0.0), t, args=(10.0, 28.0, 3.0)
track2 = odeint(lorenz, (0.0, 1.01, 0.0), t, args=(10.0, 28.0, 3.0)

# 绘图
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

fig = plt.figure()
ax = Axes3D(fig)
ax.plot(track1[:,0], track1[:,1], track1[:,2])
ax.plot(track2[:,0], track2[:,1], track2[:,2])
plt.show()
```



用odeint函数对洛仑兹吸引子微分方程进行数值求解所得到的运动轨迹

我们看到即使初始值只相差0.01，两条运动轨迹也是完全不同的。

在程序中先定义一个lorenz函数，它的任务是计算出某个位置的各个方向的微分值，这个计算直接根据洛仑兹吸引子的公式得出。然后调用odeint，对微分方程求解，odeint有许多参数，这里用到的四个参数分别为：

1. lorenz， 它是计算某个位移上的各个方向的速度(位移的微分)
2. (0.0, 1.0, 0.0)，位移初始值。计算常微分方程所需的各个变量的初始值
3. t， 表示时间的数组，odeint对于此数组中的每个时间点进行求解，得出所有时间点的位置
4. args， 这些参数直接传递给lorenz函数，因此它们都是常量

滤波器设计

scipy.signal库提供了许多信号处理方面的函数。在这一节，让我们来看看如何利用signal库设计滤波器，查看滤波器的频率响应，以及如何使用滤波器对信号进行滤波。

假设如下导入signal库:

```
>>> import scipy.signal as signal
```

下面的程序设计一个带通IIR滤波器：

```
>>> b, a = signal.iirdesign([0.2, 0.5], [0.1, 0.6], 2, 40)
```

这个滤波器的通带为0.2*f0*到*0.5*f0，阻带为小于0.1*f0*和大于*0.6*f0，其中f0为1/2的信号取样频率，如果取样频率为8kHz的话，那么这个带通滤波器的通带为800Hz到2kHz。通带的最大增益衰减为2dB，阻带的最小增益衰减为40dB，即通带的增益浮动在2dB之内，阻带至少有40dB的衰减。

iirdesgin返回的两个数组b和a， 它们分别是IIR滤波器的分子和分母部分的系数。其中a[0]恒等于1。

下面通过调用freqz计算所得到的滤波器的频率响应：

```
>>> w, h = signal.freqz(b, a)
```

freqz返回两个数组w和h，其中w是圆频率数组，通过w/pi*f0可以计算出其对应的实际频率。h是w中的对应频率点的响应，它是一个复数数组，其幅值为滤波器的增益，相角为滤波器的相位特性。

下面计算h的增益特性，并转换为dB度量。由于h中存在幅值几乎为0的值，因此先用clip函数对其裁剪之后，再调用对数函数，避免计算出错。

```
>>> power = 20*np.log10(np.clip(np.abs(h), 1e-8, 1e100))
```

通过下面的语句可以绘制出滤波器的增益特性图，这里假设取样频率为8kHz：

```
>>> pl.plot(w/np.pi*4000, power)
```

在实际运用中为了测量未知系统的频率特性，经常将频率扫描波输入到系统中，观察系统的输出，从而计算其频率特性。下面让我们来模拟这一过程。

为了调用chirp函数以产生频率扫描波形的数据，首先需要产生一个等差数组代表取样时间，下面的语句产生2秒钟取样频率为8kHz的取样时间数组：

```
>>> t = np.arange(0, 2, 1/8000.0)
```

然后调用chirp得到2秒钟的频率扫描波形的数据：

```
>>> sweep = signal.chirp(t, f0=0, t1 = 2, f1=4000.0)
```

频率扫描波的开始频率f0为0Hz，结束频率f1为4kHz，到达4kHz的时间为2秒，使用数组t作为取样时间点。

下面通过调用lfilter函数计算sweep波形经过带通滤波器之后的结果：

```
>>> out = signal.lfilter(b, a, sweep)
```

lfilter内部通过如下算式计算IIR滤波器的输出：

通过如下算式可以计算输入为x时的滤波器的输出，其中数组x代表输入信号，y代表输出信号：

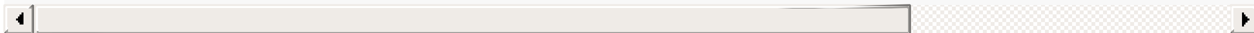
$$\begin{aligned} y[n] &= b[0]x[n] + b[1]x[n-1] + \cdots + b[P]x[n-P] \\ &\quad - a[1]y[n-1] - a[2]y[n-2] - \cdots - a[Q]y[n-Q] \end{aligned}$$

为了和系统的增益特性图进行比较，需要获取输出波形的包络，因此下面先将输出波形数据转换为能量值：

```
>>> out = 20*np.log10(np.abs(out))
```

为了计算包络，找到所有能量大于前后两个取样点(局部最大点)的下标：

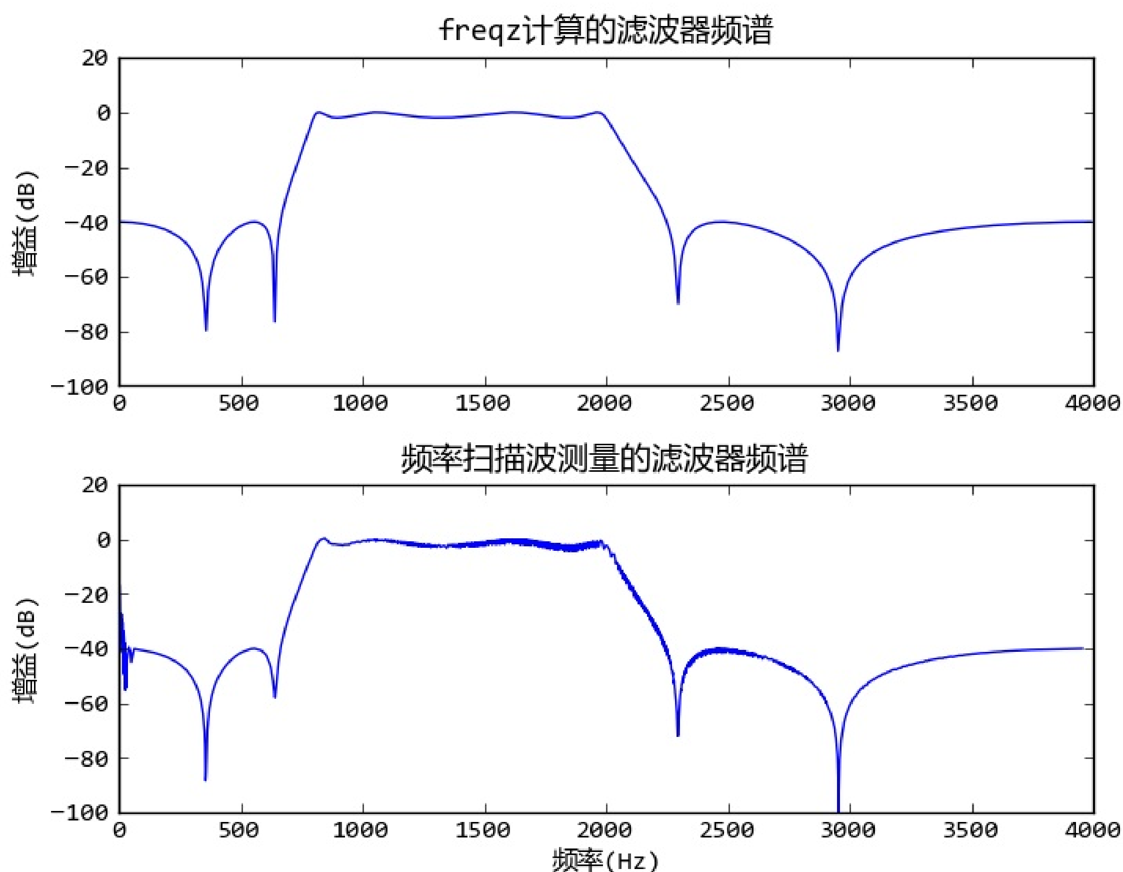
```
>>> index = np.where(np.logical_and(out[1:-1] > out[:-2], out[1:-1]
```



最后将时间转换为对应的频率，绘制所有局部最大点的能量值：

```
>>> pl.plot(t[index]/2.0*4000, out[index] )
```

下图显示freqz计算的频谱和频率扫描波得到的频率特性，我们看到其结果是一致的。



带通IIR滤波器的频率响应和频率扫描波计算的结果比较

计算此图的完整源程序请查看附录中的 带通滤波器设计 。

用Weave嵌入C语言

Python作为动态语言其功能虽然强大，但是在数值计算方面有一个最大的缺点：速度不够快。在Python级别的循环和计算的速度只有C语言程序的百分之一。因此才有了NumPy, SciPy这样的函数库，将高度优化的C、Fortran的函数库进行包装，以供Python程序调用。如果这些高度优化的函数库无法实现我们的算法，必须从头开始写循环、计算的话，那么用Python来做显然是不合适的。因此SciPy提供了快速调用C++语言程序的方法-- Weave。下面是对NumPy的数组求和的例子：

```
# -*- coding: utf-8 -*-
import scipy.weave as weave
import numpy as np
import time

def my_sum(a):
    n=int(len(a))
    code="""
 int i;

 double counter;
 counter =0;
 for(i=0;i<n;i++){
 counter=counter+a(i);
 }
 return_val=counter;
 """

    err=weave.inline(
        code,['a','n'],
        type_converters=weave.converters.blitz,
        compiler="gcc"
    )
    return err

a = np.arange(0, 10000000, 1.0)
# 先调用一次my_sum，weave会自动对C语言进行编译，此后直接运行编译之后的代码
my_sum(a)

start = time.clock()
for i in xrange(100):
    my_sum(a)  # 直接运行编译之后的代码
print "my_sum:", (time.clock() - start) / 100.0

start = time.clock()
for i in xrange(100):
    np.sum( a ) # numpy中的sum，其实现也是C语言级别
print "np.sum:", (time.clock() - start) / 100.0

start = time.clock()
print sum(a) # Python内部函数sum通过数组a的迭代接口访问其每个元素，因此速度
print "sum:", time.clock() - start
```

此例子在我的电脑上的运行结果为：

```
>>> from sympy import *
```

封面上的经典公式

本书的封面上的公式：

$$e^{i\pi}+1=0$$

叫做欧拉恒等式，其中e是自然指数的底，i是虚数单位， π 是圆周率。此公式被誉为数学最奇妙的公式，它将5个基本数学常数用加法、乘法和幂运算联系起来。下面用SymPy验证一下这个公式。

载入的符号中，E表示自然指数的底，I表示虚数单位，pi表示圆周率，因此上述的公式可以直接如下计算：

```
>>> E**(I*pi)+1
0
```

欧拉恒等式可以下面的公式进行计算，

$$e^{ix}=\cos x+i\sin x$$

为了用SymPy求证上面的公式，我们需要引入变量x。在SymPy中，数学符号是Symbol类的对象，因此必须先创建之后才能使用：

```
>>> x = Symbol('x')
```

expand函数可以将公式展开，我们用它来展开E*(/pi)试试看：

```
>>> expand( E**(I*x) )
exp(I*x)
```

没有成功，只是换了一种写法而已。这里的exp不是math.exp或者numpy.exp，而是sympy.exp，它是一个类，用来表述自然指数函数。

expand函数有关键字参数complex，当它为True时，expand将把公式分为实数和虚数两个部分：

```
>>> expand(exp(I*x), complex=True)
I*exp(-im(x))*sin(re(x)) + cos(re(x))*exp(-im(x))
```

这次得到的结果相当复杂，其中sin, cos, re, im都是sympy定义的类，re表示取实数部分，im表示取虚数部分。显然这里的运算将符号x当作复数了。为了指定符号x必须是实数，我们需要如下重新定义符号x：

```
>>> x = Symbol("x", real=True)
>>> expand(exp(I*x), complex=True)
I*sin(x) + cos(x)
```

终于得到了我们需要的公式。那么如何证明它呢。我们可以用泰勒多项式展开：

```
>>> tmp = series(exp(I*x), x, 0, 10)
>>> pprint(tmp)
 2      3     4      5      6      7      8        9
 x    I*x     x    I*x      x    I*x      x      I*x
1 + I*x - -- - ---- + -- + ---- - --- - ---- + ----- + ------ + O(x
 2     6     24   120    720   5040   40320   362880
```

series是泰勒展开函数，pprint将公式用更好看的格式打印出来。下面分别获得tmp的实部和虚部，分别和cos(x)和sin(x)的展开公式进行比较：

```
&gt;&gt;&gt; pprint(re(tmp))
 2    4     6      8
 x    x     x      x
1 + re(O(x**10)) - -- + -- - --- + -----
 2    24   720   40320
```

```
&gt;&gt;&gt; pprint( series( cos(x), x, 0, 10) )
 2    4     6      8
 x    x     x      x
1 - -- + -- - --- + ----- + O(x**10)
 2    24   720   40320
```

```
&gt;&gt;&gt; pprint(im(tmp))
 3     5     7       9
 x     x     x       x
x + im(O(x**10)) - -- + --- - ---- + ------
 6    120   5040   362880
```

```
&gt;&gt;&gt; pprint(series(sin(x), x, 0, 10))
 3     5     7       9
 x     x     x       x
x - -- + --- - ---- + ------ + O(x**10)
 6    120   5040   362880
```

球体体积

在用SciPy数值积分一节我们介绍了如何使用数值定积分计算球体的体积，而SymPy的符号积分函数integrate则可以帮助我们进行符号积分。integrate可以进行不定积分：

```
>>> integrate(x*sin(x), x)
-x*cos(x) + sin(x)
```

如果指定x的取值范围的话，integrate则进行定积分运算：

```
>>> integrate(x*sin(x), (x, 0, 2*pi))
-2*pi
```

为了计算球体体积，首先让我们来看看如何计算圆形面积，假设圆形的半径为r，则圆上任意一点的Y坐标函数为：

$$y(x) = \sqrt{r^2 - x^2}$$

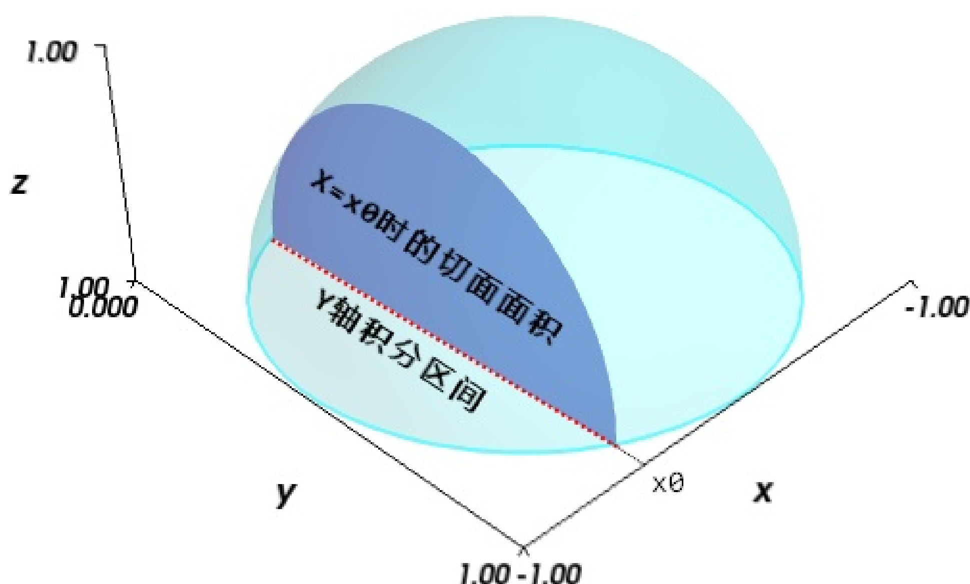
因此我们可以直接对上述函数在-r到r区间上进行积分得到半圆面积，注意这里我们使用symbols函数一次创建多个符号：

```
>>> x, y, r = symbols('x,y,r')
>>> 2 * integrate(sqrt(r*r-x**2), (x, -r, r))
2*Integral((r**2 - x**2)**(1/2), (x, -r, r))
```

很遗憾，integrate函数没有计算出结果，而是直接返回了我们输入的算式。这是因为SymPy不知道r是大于0的，如下重新定义r，就可以得到正确答案了：

```
>>> r = symbols('r', positive=True)
>>> circle_area = 2 * integrate(sqrt(r**2-x**2), (x, -r, r))
>>> circle_area
pi*r**2
```

接下来对此面积公式进行定积分，就可以得到球体的体积，但是随着X轴坐标的变化，对应的切面的的半径会发生变化，现在假设X轴的坐标为x，球体的半径为r，则x处的切面的半径为可以使用前面的公式y(x)计算出。



球体体积的双重定积分示意图

因此我们需要对circle_area中的变量r进行替代：

```
>>> circle_area = circle_area.subs(r, sqrt(r**2-x**2))
>>> circle_area
pi*(r**2 - x**2)
```

用subs进行算式替换

subs函数可以将算式中的符号进行替换，它有3种调用方式：

- expression.subs(x, y) : 将算式中的x替换成y
- expression.subs({x:y,u:v}) : 使用字典进行多次替换
- expression.subs([(x,y),(u,v)]) : 使用列表进行多次替换

请注意多次替换是顺序执行的，因此：

```
expression.sub([(x,y),(y,x)])
```

并不能对两个符号x,y进行交换。

然后对circle_area中的变量x在区间-r到r上进行定积分，得到球体的体积公式：

```
>>> integrate(circle_area, (x, -r, r))
4*pi*r**3/3
```

matplotlib-绘制精美的图表

matplotlib 是python最著名的绘图库，它提供了一整套和matlab相似的命令API，十分适合交互式地进行制图。而且也可以方便地将它作为绘图控件，嵌入GUI应用程序中。

它的文档相当完备，并且 Gallery页面 中有上百幅缩略图，打开之后都有源程序。因此如果你需要绘制某种类型的图，只需要在这个页面中浏览/复制/粘贴一下，基本上都能搞定。

本章节作为matplotlib的入门介绍，将较为深入地挖掘几个例子，从中理解和学习matplotlib绘图的一些基本概念。

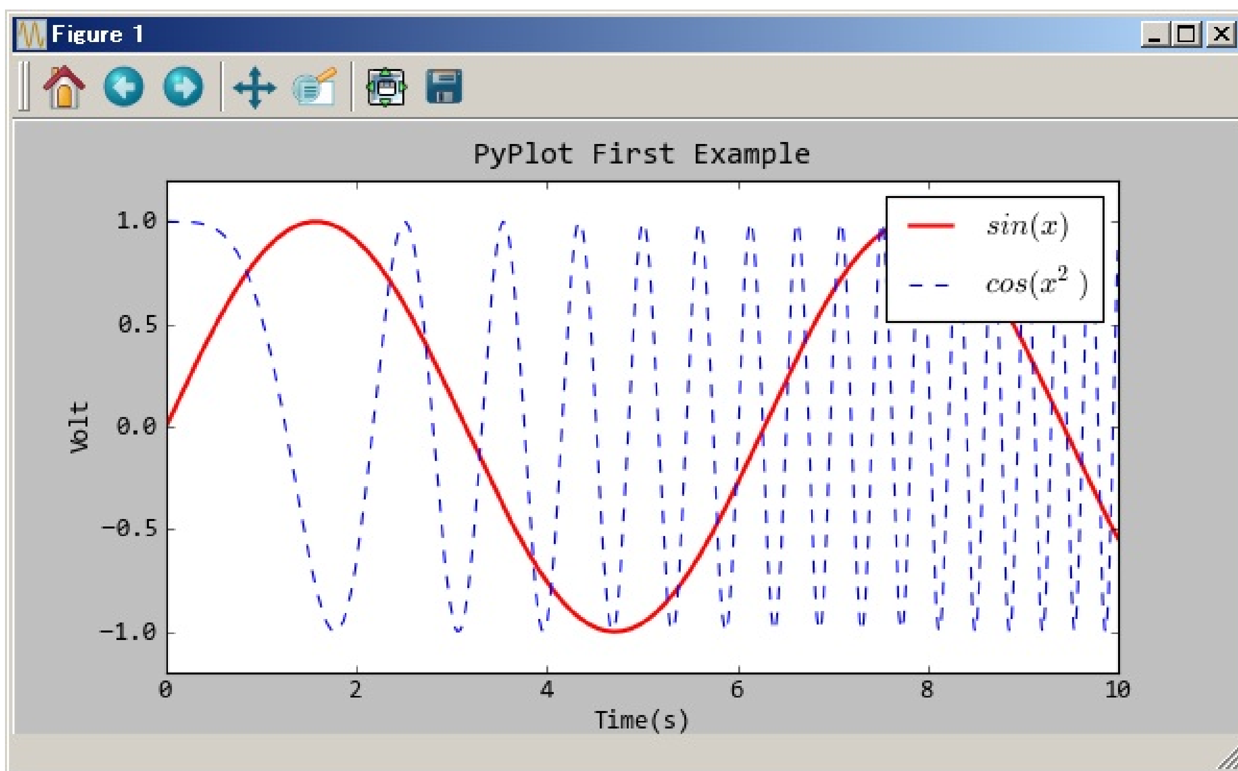
快速绘图

matplotlib的pyplot子库提供了和matlab类似的绘图API，方便用户快速绘制2D图表。让我们先来看一个简单的例子：

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 1000)
y = np.sin(x)
z = np.cos(x**2)

plt.figure(figsize=(8,4))
plt.plot(x,y,label="$sin(x)$",color="red",linewidth=2)
plt.plot(x,z,"b--",label="$cos(x^2)$")
plt.xlabel("Time(s)")
plt.ylabel("Volt")
plt.title("PyPlot First Example")
plt.ylim(-1.2,1.2)
plt.legend()
plt.show()
```



调用pyplot库快速将数据绘制成曲线图

matplotlib中的快速绘图的函数库可以通过如下语句载入：

```
import matplotlib.pyplot as plt
```

pylab模块

matplotlib还提供了名为pylab的模块，其中包括了许多numpy和pyplot中常用的函数，方便用户快速进行计算和绘图，可以用于IPython中的快速交互式使用。

接下来调用figure创建一个绘图对象，并且使它成为当前的绘图对象。

```
plt.figure(figsize=(8,4))
```

也可以不创建绘图对象直接调用接下来的plot函数直接绘图，matplotlib会为我们自动创建一个绘图对象。如果需要同时绘制多幅图表的话，可以是给figure传递一个整数参数指定图标的序号，如果所指定序号的绘图对象已经存在的话，将不创建新的对象，而只是让它成为当前绘图对象。

通过figsize参数可以指定绘图对象的宽度和高度，单位为英寸；dpi参数指定绘图对象的分辨率，即每英寸多少个像素，缺省值为80。因此本例中所创建的图表窗口的宽度为8*80 = 640像素。

但是用工具栏中的保存按钮保存下来的png图像的大小是800*400像素。这是因为保存图表用的函数savefig使用不同的DPI配置，savefig函数也有一个dpi参数，如果不设置的话，将使用matplotlib配置文件中的配置，此配置可以通过如下语句进行查看，关于配置文件将在后面的章节进行介绍：

```
>>> import matplotlib
>>> matplotlib.rcParams["savefig.dpi"]
100
```

下面的两行程序通过调用plot函数在当前的绘图对象中进行绘图：

```
plt.plot(x,y,label="$sin(x)$",color="red",linewidth=2)
plt.plot(x,z,"b--",label="$cos(x^2)$")
```

plot函数的调用方式很灵活，第一句将x,y数组传递给plot之后，用关键字参数指定各种属性：

- **label** : 给所绘制的曲线一个名字，此名字在图示(legend)中显示。只要在字符串前后添加"$"符号，matplotlib就会使用其内嵌的latex引擎绘制的数学公式。
- **color** : 指定曲线的颜色
- **linewidth** : 指定曲线的宽度

第二句直接通过第三个参数"b--"指定曲线的颜色和线型，这个参数称为格式化参数，它能够通过一些易记的符号快速指定曲线的样式。其中b表示蓝色，"--"表示线型为虚线。在IPython中输入 "plt.plot?" 可以查看格式化字符串的详细配置。

接下来通过一系列函数设置绘图对象的各个属性：

```
plt.xlabel("Time(s)")
plt.ylabel("Volt")
plt.title("PyPlot First Example")
plt.ylim(-1.2,1.2)
plt.legend()
```

- **xlabel** : 设置X轴的文字
- **ylabel** : 设置Y轴的文字
- **title** : 设置图表的标题
- **ylim** : 设置Y轴的范围
- **legend** : 显示图示

最后调用plt.show()显示出我们创建的所有绘图对象。

配置属性

matplotlib所绘制的图的每个组成部分都对应有一个对象，我们可以通过调用这些对象的属性设置方法set_*或者pyplot的属性设置函数setp设置其属性值。例如plot函数返回一个 matplotlib.lines.Line2D 对象的列表，下面的例子显示如何设置Line2D对象的属性：

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.arange(0, 5, 0.1)
>>> line, = plt.plot(x, x*x) # plot返回一个列表，通过line,获取其第一个元
>>> # 调用Line2D对象的set_*方法设置属性值
>>> line.set_antialiased(False)
```

```
>>> # 同时绘制sin和cos两条曲线，lines是一个有两个Line2D对象的列表
>>> lines = plt.plot(x, np.sin(x), x, np.cos(x)) #
>>> # 调用setp函数同时配置多个Line2D对象的多个属性值
>>> plt.setp(lines, color="r", linewidth=2.0)
```

这段例子中，通过调用Line2D对象line的set_antialiased方法，关闭对象的反锯齿效果。或者通过调用plt.setp函数配置多个Line2D对象的颜色和线宽属性。

同样我们可以通过调用Line2D对象的get_*方法，或者plt.getp函数获取对象的属性值：

```
>>> line.get_linewidth()
1.0
>>> plt.getp(lines[0], "color") # 返回color属性
'r'
>>> plt.getp(lines[1]) # 输出全部属性
alpha = 1.0
animated = False
antialiased or aa = True
axes = Axes(0.125,0.1;0.775x0.8)
... ...
```

注意getp函数只能对一个对象进行操作，它有两种用法：

- 指定属性名：返回对象的指定属性的值
- 不指定属性名：打印出对象的所有属性和其值

matplotlib的整个图表为一个Figure对象，此对象在调用plt.figure函数时返回，我们也可以通过plt.gcf函数获取当前的绘图对象：

```
>>> f = plt.gcf()
>>> plt.getp(f)
alpha = 1.0
animated = False
...
```

Figure对象有一个axes属性，其值为AxesSubplot对象的列表，每个AxesSubplot对象代表图表中的一个子图，前面所绘制的图表只包含一个子图，当前子图也可以通过plt.gca获得：

```
>>> plt.getp(f, "axes")
[<matplotlib.axes.AxesSubplot object at 0x05CDD170>]
>>> plt.gca()
<matplotlib.axes.AxesSubplot object at 0x05CDD170>
```

用plt.getp可以发现AxesSubplot对象有很多属性，例如它的lines属性为此子图所包括的 Line2D 对象列表：

```
>>> alllines = plt.getp(plt.gca(), "lines")
>>> alllines
<a list of 3 Line2D objects>
>>> alllines[0] == line # 其中的第一条曲线就是最开始绘制的那条曲线
True
```

通过这种方法我们可以很容易地查看对象的属性和它们之间的包含关系，找到需要配置的属性。

绘制多轴图

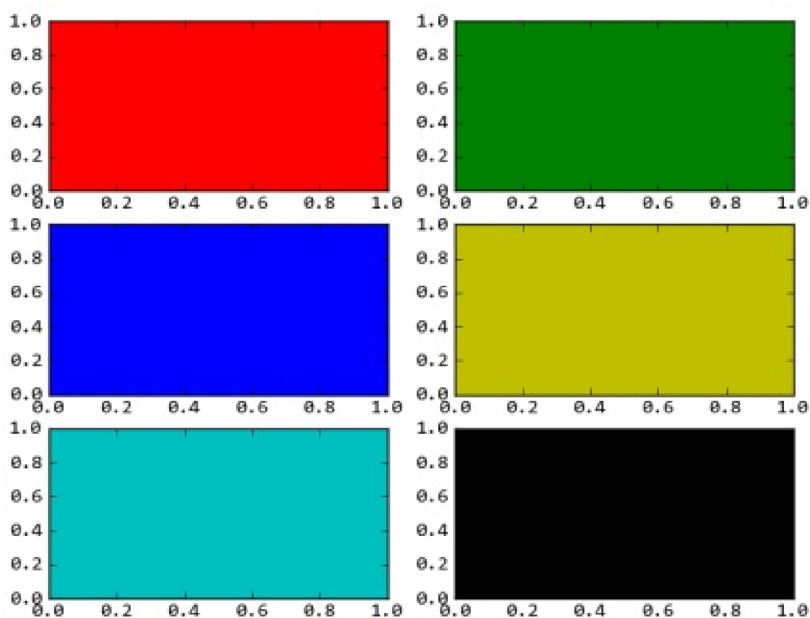
一个绘图对象(figure)可以包含多个轴(axis)，在Matplotlib中用轴表示一个绘图区域，可以将其理解为子图。上面的第一个例子中，绘图对象只包括一个轴，因此只显示了一个轴(子图)。我们可以使用subplot函数快速绘制有多个轴的图表。subplot函数的调用形式如下：

```
subplot(numRows, numCols, plotNum)
```

subplot将整个绘图区域等分为numRows行 * numCols列个子区域，然后按照从左到右，从上到下的顺序对每个子区域进行编号，左上的子区域的编号为1。如果numRows，numCols和plotNum这三个数都小于10的话，可以把它们缩写为一个整数，例如subplot(323)和subplot(3,2,3)是相同的。subplot在plotNum指定的区域中创建一个轴对象。如果新创建的轴和之前创建的轴重叠的话，之前的轴将被删除。

下面的程序创建3行2列共6个轴，通过axisbg参数给每个轴设置不同的背景颜色。

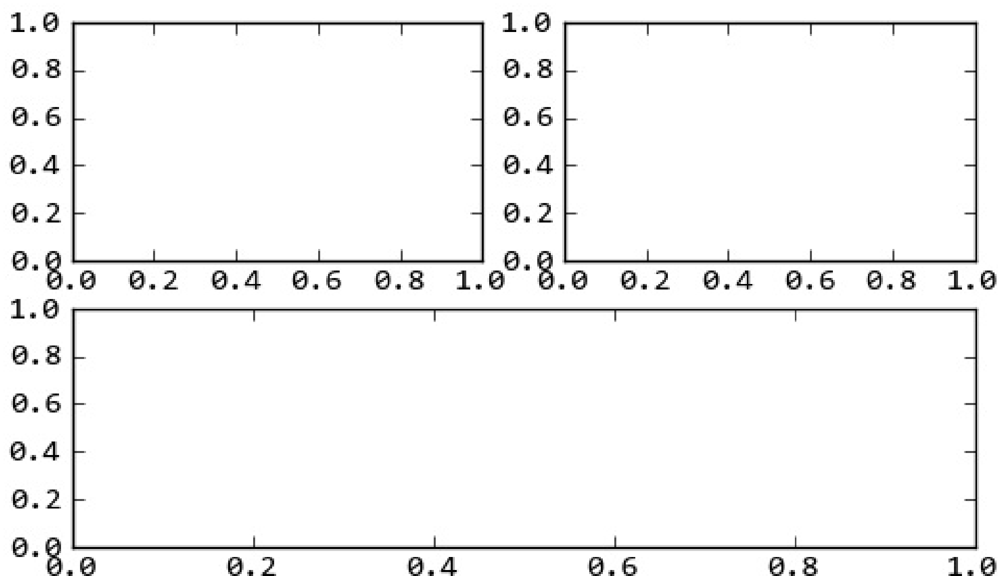
```
for idx, color in enumerate("rgbyck"):
    plt.subplot(320+idx+1, axisbg=color)
plt.show()
```



用subplot函数将Figure分为六个子图区域

如果希望某个轴占据整个行或者列的话，可以如下调用subplot：

```
plt.subplot(221) # 第一行的左图
plt.subplot(222) # 第一行的右图
plt.subplot(212) # 第二整行
plt.show()
```



将Figure分为三个子图区域

当绘图对象中有多个轴的时候，可以通过工具栏中的Configure Subplots按钮，交互式地调节轴之间的间距和轴与边框之间的距离。如果希望在程序中调节的话，可以调用subplots_adjust函数，它有left, right, bottom, top, wspace, hspace等几个关

键字参数，这些参数的值都是0到1之间的小数，它们是以绘图区域的宽高为1进行正规化之后的坐标或者长度。

配置文件

一幅图有许多需要配置的属性，例如颜色、字体、线型等等。我们在绘图时，并没有一一对这些属性进行配置，许多都直接采用了Matplotlib的缺省配置。Matplotlib将缺省配置保存在一个文件中，通过更改这个文件，我们可以修改这些属性的缺省值。

Matplotlib 使用配置文件 matplotlibrc 时的搜索顺序如下：

- 当前路径：程序的当前路径
- 用户配置路径：通常为 HOME/.matplotlib/，可以通过环境变量 MATPLOTLIBRC修改
- 系统配置路径：保存在 matplotlib的安装目录下的 mpl-data 下

通过下面的语句可以获取用户配置路径：

```
>>> import matplotlib
>>> matplotlib.get_configdir()
'C:\\Documents and Settings\\zhang\\.matplotlib'
```

通过下面的语句可以获得目前使用的配置文件的路径：

```
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
'C:\\Python26\\lib\\site-packages\\matplotlib\\mpl-data\\matplotlib
```

由于在当前路径和用户配置路径中都没有找到位置文件，因此最后使用的是系统配置路径下的配置文件。如果你将matplotlibrc复制一份到脚本的当前目录下：

```
>>> import os
>>> os.getcwd()
'C:\\zhang\\doc'
```

复制配置文件之后再运行:

```
>>> matplotlib.matplotlib_fname()
'C:\\zhang\\doc\\matplotlibrc'
```

如果你用文本编辑器打开此配置文件的话，你会发现它实际上是定义了一个字典。为了对众多的配置进行区分，关键字可以用点分开。

配置文件的读入可以使用 rc_params 函数，它返回一个配置字典：

```
>>> matplotlib.rc_params()
{'agg.path.chunksize': 0,
 'axes.axisbelow': False,
 'axes.edgecolor': 'k',
 'axes.facecolor': 'w',
 ... ...
```

在matplotlib模块载入的时候会调用rc_params，并把得到的配置字典保存到rcParams变量中：

```
>>> matplotlib.rcParams
{'agg.path.chunksize': 0,
'axes.axisbelow': False,
... ...
```

matplotlib将使用rcParams中的配置进行绘图。用户可以直接修改此字典中的配置，所做的改变会反映到此后所绘制的图中。例如下面的脚本所绘制的线将带有圆形的点标识符：

```
>>> matplotlib.rcParams["lines.marker"] = "o"
>>> import pylab
>>> pylab.plot([1,2,3])
>>> pylab.show()
```

为了方便配置，可以使用rc函数，下面的例子同时配置点标识符、线宽和颜色：

```
>>> matplotlib.rc("lines", marker="x", linewidth=2, color="red")
```

如果希望恢复到缺省的配置(matplotlib载入时从配置文件读入的配置)的话，可以调用 rcdefaults 函数。

```
>>> matplotlib.rcdefaults()
```

如果手工修改了配置文件，希望重新从配置文件载入最新的配置的话，可以调用：

```
>>> matplotlib.rcParams.update( matplotlib.rc_params() )
```

Artist对象

matplotlib API包含有三层：

- **backend_bases.FigureCanvas** : 图表的绘制领域
- **backend_bases.Renderer** : 知道如何在FigureCanvas上如何绘图
- **artist.Artist** : 知道如何使用Renderer在FigureCanvas上绘图

FigureCanvas和Renderer需要处理底层的绘图操作，例如使用wxPython在界面上绘图，或者使用PostScript绘制PDF。Artist则处理所有的高层结构，例如处理图表、文字和曲线等的绘制和布局。通常我们只和Artist打交道，而不需要关心底层的绘制细节。

Artists分为简单类型和容器类型两种。简单类型的Artists为标准的绘图元件，例如Line2D、 Rectangle、 Text、AxesImage 等等。而容器类型则可以包含许多简单类型的Artists，使它们组织成一个整体，例如Axis、 Axes、Figure等。

直接使用Artists创建图表的标准流程如下：

- 创建Figure对象
- 用Figure对象创建一个或者多个Axes或者Subplot对象
- 调用Axies等对象的方法创建各种简单类型的Artists

下面首先调用pyplot.figure辅助函数创建Figure对象，然后调用Figure对象的add_axes方法在其中创建一个Axes对象，add_axes的参数是一个形如[left, bottom, width, height]的列表，这些数值分别指定所创建的Axes对象相对于fig的位置和大小，取值范围都在0到1之间：

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = fig.add_axes([0.15, 0.1, 0.7, 0.3])
```

然后我们调用ax的plot方法绘图，创建一条曲线，并且返回此曲线对象(Line2D)。

```
>>> line, = ax.plot([1,2,3],[1,2,1])
>>> ax.lines
[<matplotlib.lines.Line2D object at 0x0637A3D0>]
>>> line
<matplotlib.lines.Line2D object at 0x0637A3D0>
```

ax.lines是一个为包含ax的所有曲线的列表，后续的ax.plot调用会往此列表中添加新的曲线。如果想删除某条曲线的话，直接从此列表中删除即可。

Axes对象还包括许多其它的Artists对象，例如我们可以通过调用set_xlabel设置其X轴上的标题：

```
>>> ax.set_xlabel("time")
```

如果我们查看set_xlabel的源代码的话，会发现它是通过调用下面的语句实现的：

```
self.xaxis.set_label_text(xlabel)
```

如果我们一直跟踪下去，会发现Axes的xaxis属性是一个XAxis对象：

```
>>> ax.xaxis
<matplotlib.axis.XAxis object at 0x06343230>
```

XAxis的label属性是一个Text对象：

```
>>> ax.xaxis.label
<matplotlib.text.Text object at 0x06343290>
```

而Text对象的_text属性为我们设置的值：

```
>>> ax.xaxis.label._text
'time'
```

这些对象都是Artists，因此也可以调用它们的属性获取函数来获得相应的属性：

```
>>> ax.xaxis.label.get_text()
'time'
```

Artist的属性

图表中的每个元素都用一个matplotlib的Artist对象表示，而每个Artist对象都有一大堆属性控制其显示效果。例如Figure对象和Axes对象都有patch属性作为其背景，它的值是一个Rectangle对象。通过设置此它的一些属性可以修改Figrue图表的背景颜色或者透明度等属性，下面的例子将图表的背景颜色设置为绿色：

```
>>> fig = plt.figure()
>>> fig.show()
>>> fig.patch.set_color("g")
>>> fig.canvas.draw()
```

patch的color属性通过set_color函数进行设置，属性修改之后并不会立即反映到图表的显示上，还需要调用fig.canvas.draw()函数才能够更新显示。

下面是Artist对象都具有的一些属性：

- alpha : 透明度，值在0到1之间，0为完全透明，1为完全不透明
- animated : 布尔值，在绘制动画效果时使用
- axes : 此Artist对象所在的Axes对象，可能为None

- clip_box : 对象的裁剪框
- clip_on : 是否裁剪
- clip_path : 裁剪的路径
- contains : 判断指定点是否在对象上的函数
- figure : 所在的Figure对象，可能为None
- label : 文本标签
- picker : 控制Artist对象选取
- transform : 控制偏移旋转
- visible : 是否可见
- zorder : 控制绘图顺序

Artist对象的所有属性都通过相应的 get* 和 *set** 函数进行读写，例如下面的语句将alpha属性设置为当前值的一半：

```
>>> fig.set_alpha(0.5*fig.get_alpha())
```

如果你想用一条语句设置多个属性的话，可以使用set函数：

```
>>> fig.set(alpha=0.5, zorder=2)
```

使用前面介绍的 matplotlib.pyplot.getp 函数可以方便地输出Artist对象的所有属性名和值。

```
>>> plt.getp(fig.patch)
 aa = True
 alpha = 1.0
 animated = False
 antialiased or aa = True
 ... ...
```

Figure容器

现在我们知道如何观察和修改已知的某个Artist对象的属性，接下来要解决如何找到指定的Artist对象。前面我们介绍过Artist对象有容器类型和简单类型两种，这一节让我们来详细看看容器类型的内容。

最大的Artist容器是matplotlib.figure.Figure，它包括组成图表的所有元素。图表的背景是一个Rectangle对象，用Figure.patch属性表示。当你通过调用add_subplot或者add_axes方法往图表中添加轴(子图时)，这些子图都将添加到Figure.axes属性中，同时这两个方法也返回添加进axes属性的对象，注意返回值的类型有所不同，实际上AxesSubplot是Axes的子类。

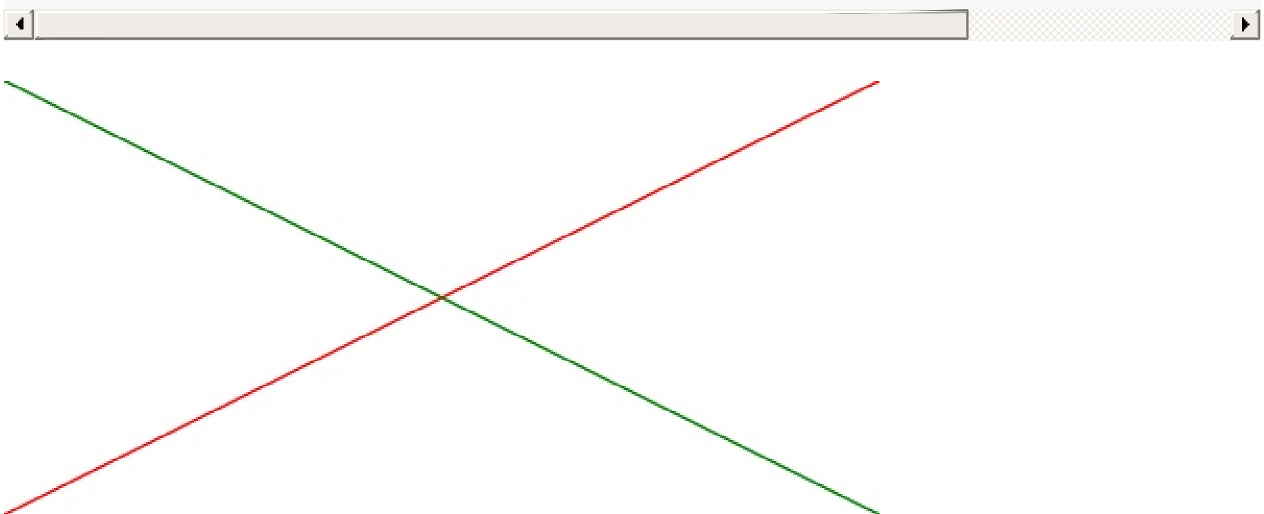
```
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(211)
>>> ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])
>>> ax1
<matplotlib.axes.AxesSubplot object at 0x056BCA90>
>>> ax2
<matplotlib.axes.Axes object at 0x056BC910>
>>> fig.axes
[<matplotlib.axes.AxesSubplot object at 0x056BCA90>,
<matplotlib.axes.Axes object at 0x056BC910>]
```

为了支持pylab中的gca()等函数，Figure对象内部保存有当前轴的信息，因此不建议直接对Figure.axes属性进行列表操作，而应该使用add_subplot, add_axes, delaxes等方法进行添加和删除操作。但是使用for循环对axes中的每个元素进行操作是没有问题的，下面的语句打开所有子图的栅格。

```
>>> for ax in fig.axes: ax.grid(True)
```

Figure对象可以拥有自己的文字、线条以及图像等简单类型的Artist。缺省的坐标系统为像素点，但是可以通过设置Artist对象的transform属性修改坐标系的转换方式。最常用的Figure对象的坐标系是以左下角为坐标原点(0,0)，右上角为坐标(1,1)。下面的程序创建并添加两条直线到fig中：

```
>>> from matplotlib.lines import Line2D
>>> fig = plt.figure()
>>> line1 = Line2D([0,1],[0,1], transform=fig.transFigure, figure=f
>>> line2 = Line2D([0,1],[1,0], transform=fig.transFigure, figure=f
>>> fig.lines.extend([line1, line2])
>>> fig.show()
```



在Figure对象中手工绘制直线

注意为了让所创建的Line2D对象使用fig的坐标，我们将fig.TransFigure赋给Line2D对象的transform属性；为了让Line2D对象知道它是在fig对象中，我们还设置其figure属性为fig；最后还需要将创建的两个Line2D对象添加到fig.lines属性中去。

Figure对象有如下属性包含其它的Artist对象：

- axes : Axes对象列表
- patch : 作为背景的Rectangle对象
- images : FigureImage对象列表，用来显示图片
- legends : Legend对象列表
- lines : Line2D对象列表
- patches : patch对象列表
- texts : Text对象列表，用来显示文字

Axes容器

Axes容器是整个matplotlib库的核心，它包含了组成图表的众多Artist对象，并且有许多方法函数帮助我们创建、修改这些对象。和Figure一样，它有一个patch属性作为背景，当它是笛卡尔坐标时，patch属性是一个Rectangle对象，而当它是极坐标时，patch属性则是Circle对象。例如下面的语句设置Axes对象的背景颜色为绿色：

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.patch.set_facecolor("green")
```

当你调用Axes的绘图方法（例如plot），它将创建一组Line2D对象，并将所有的关键字参数传递给这些Line2D对象，并将它们添加进Axes.lines属性中，最后返回所创建的Line2D对象列表：

```
>>> x, y = np.random.rand(2, 100)
>>> line, = ax.plot(x, y, "-", color="blue", linewidth=2)
>>> line
<matplotlib.lines.Line2D object at 0x03007030>
>>> ax.lines
[<matplotlib.lines.Line2D object at 0x03007030>]
```

注意plot返回的是一个Line2D对象的列表，因为我们可以传递多组X,Y轴的数据，一次绘制多条曲线。

与plot方法类似，绘制直方图的方法bar和绘制柱状统计图的方法hist将创建一个Patch对象的列表，每个元素实际上都是Patch的子类Rectangle，并且将所创建的Patch对象都添加进Axes.patches属性中：

```
>>> ax = fig.add_subplot(111)
>>> n, bins, rects = ax.hist(np.random.randn(1000), 50, facecolor='
>>> rects
<a list of 50 Patch objects>
>>> rects[0]
<matplotlib.patches.Rectangle object at 0x05BC2350>
>>> ax.patches[0]
<matplotlib.patches.Rectangle object at 0x05BC2350>
```

一般我们不会直接对Axes.lines或者Axes.patches属性进行操作，而是调用add_line或者add_patch等方法，这些方法帮助我们完成许多属性设置工作：

```
&gt;&gt;&gt; fig = plt.figure()
&gt;&gt;&gt; ax = fig.add_subplot(111)
&gt;&gt;&gt; rect = matplotlib.patches.Rectangle((1,1), width=5
&gt;&gt;&gt; print rect.get_axes() # rect的axes属性为空
None
&gt;&gt;&gt; rect.get_transform() # rect的transform属性为缺省值
BboxTransformTo(Bbox(array([[  1.,   1.],
 [  6.,  13.]])))
&gt;&gt;&gt; ax.add_patch(rect) # 将rect添加进ax
&lt;matplotlib.patches.Rectangle object at 0x05C34E50&gt;
&gt;&gt;&gt; rect.get_axes() # 于是rect的axes属性就是ax
&lt;matplotlib.axes.AxesSubplot object at 0x05C09CB0&gt;
```

```
&gt;&gt;&gt; # rect的transform属性和ax的transData相同
&gt;&gt;&gt; rect.get_transform()
... # 太长，省略
&gt;&gt;&gt; ax.transData
... # 太长，省略
```

```
&gt;&gt;&gt; ax.get_xlim() # ax的X轴范围为0到1，无法显示完整的rect
(0.0, 1.0)
&gt;&gt;&gt; ax.dataLim._get_bounds() # 数据的范围和rect的大小一致
(1.0, 1.0, 5.0, 12.0)
&gt;&gt;&gt; ax.autoscale_view() # 自动调整坐标轴范围
&gt;&gt;&gt; ax.get_xlim() # 于是X轴可以完整显示rect
(1.0, 6.0)
&gt;&gt;&gt; plt.show()
```

通过上面的例子我们可以看出，add_patch方法帮助我们设置了rect的axes和transform属性。

下面详细列出Axes包含各种Artist对象的属性：

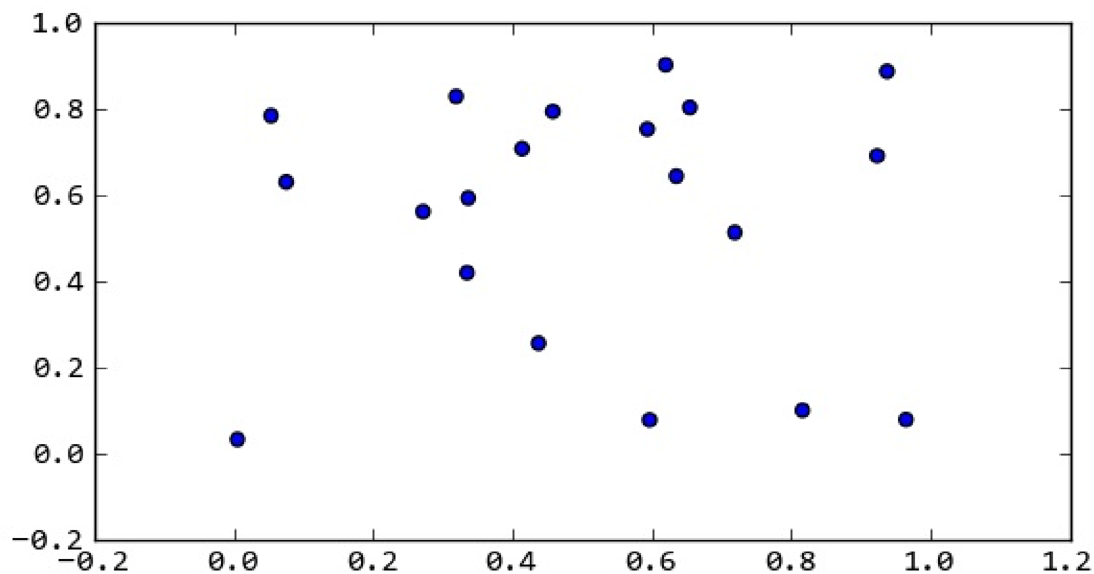
- artists : Artist对象列表
- patch : 作为Axes背景的Patch对象，可以是Rectangle或者Circle
- collections : Collection对象列表
- images : AxesImage对象列表
- legends : Legend对象列表
- lines : Line2D对象列表
- patches : Patch对象列表
- texts : Text对象列表
- xaxis : XAxis对象
- yaxis : YAxis对象

下面列出Axes的创建Artist对象的方法：

Axes的方法	所创建的对象	添加进的列表
annotate	Annotate	texts
bars	Rectangle	patches
errorbar	Line2D, Rectangle	lines,patches
fill	Polygon	patches
hist	Rectangle	patches
imshow	AxesImage	images
legend	Legend	legends
plot	Line2D	lines
scatter	PolygonCollection	Collections
text	Text	texts

下面以绘制散列图(scatter)为例，验证一下：

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> t = ax.scatter(np.random.rand(20), np.random.rand(20))
>>> t # 返回值为CircleCollection对象
<matplotlib.collections.CircleCollection object at 0x06004230>
>>> ax.collections # 返回的对象已经添加进了collections列表中
[<matplotlib.collections.CircleCollection object at 0x06004230>]
>>> fig.show()
>>> t.get_sizes() # 获得Collection的点数
20
```



用scatter函数绘制散列图

Axis容器

Axis容器包括坐标轴上的刻度线、刻度文本、坐标网格以及坐标轴标题等内容。刻度包括主刻度和副刻度，分别通过Axis.get_major_ticks和Axis.get_minor_ticks方法获得。每个刻度线都是一个XTick或者YTick对象，它包括实际的刻度线和刻度文本。为了方便访问刻度线和文本，Axis对象提供了get_ticklabels和get_ticklines方法分别直接获得刻度线和刻度文本：

```
&gt;&gt;&gt; pl.plot([1,2,3],[4,5,6])
[&lt;matplotlib.lines.Line2D object at 0x0AD3B670&gt;]
&gt;&gt;&gt; pl.show()
&gt;&gt;&gt; axis = pl.gca().xaxis
```

```
>>> axis.get_ticklocs() # 获得刻度的位置列表
array([ 1\. ,  1.5,  2\. ,  2.5,  3\. ])
```

```
>>> axis.get_ticklabels() # 获得刻度标签列表
<a list of 5 Text major ticklabel objects>
>>> [x.get_text() for x in axis.get_ticklabels()] # 获得刻度的文本字符
[u'1.0', u'1.5', u'2.0', u'2.5', u'3.0']
```

```
>>> axis.get_ticklines() # 获得主刻度线列表，图的上下刻度线共10条
<a list of 10 Line2D ticklines objects>
```

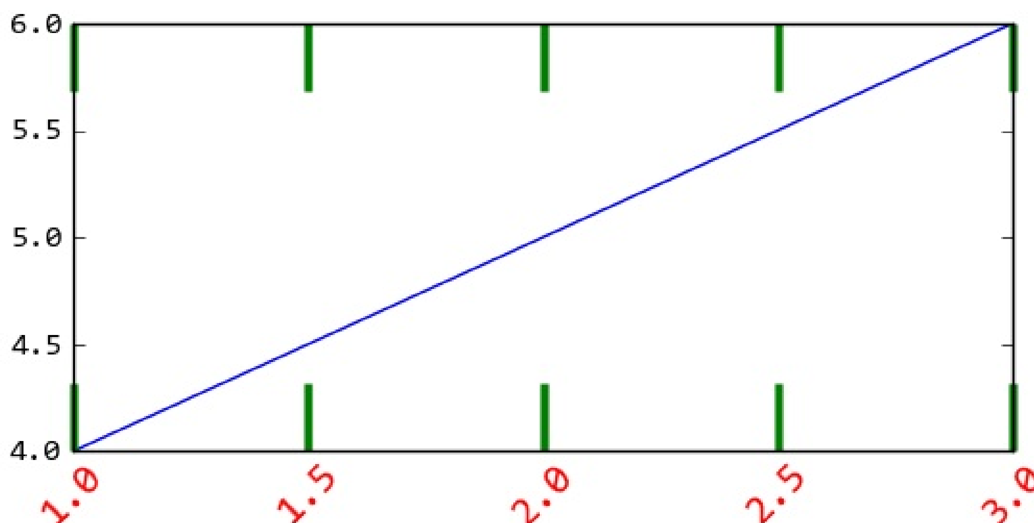
```
>>> axis.get_ticklines(minor=True) # 获得副刻度线列表
<a list of 0 Line2D ticklines objects>
```

获得刻度线或者刻度标签之后，可以设置其各种属性，下面设置刻度线为绿色粗线，文本为红色并且旋转45度：

```
&gt;&gt;&gt; for label in axis.get_ticklabels():
...     label.set_color("red")
...     label.set_rotation(45)
...     label.set_fontsize(16)
...
```

```
&gt;&gt;&gt; for line in axis.get_ticklines():
...     line.set_color("green")
...     line.set_markersize(25)
...     line.set_markeredgewidth(3)
```

最终的结果图如下：



手工配置X轴的刻度线和刻度文本的样式

上面的例子中，获得的副刻度线列表为空，这是因为用于计算副刻度的对象缺省为NullLocator，它不产生任何刻度线；而计算主刻度的对象为AutoLocator，它会根据当前的缩放等配置自动计算刻度的位置：

```
>>> axis.get_minor_locator() # 计算副刻度的对象
<matplotlib.ticker.NullLocator instance at 0x0A014300>
>>> axis.get_major_locator() # 计算主刻度的对象
<matplotlib.ticker.AutoLocator instance at 0x09281B20>
```

我们可以使用程序为Axis对象设置不同的Locator对象，用来手工设置刻度的位置；设置Formatter对象用来控制刻度文本的显示。下面的程序设置X轴的主刻度为pi/4，副刻度为pi/20，并且主刻度上的文本以pi为单位：

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as pl
from matplotlib.ticker import MultipleLocator, FuncFormatter
import numpy as np
x = np.arange(0, 4*np.pi, 0.01)
y = np.sin(x)
pl.figure(figsize=(8,4))
pl.plot(x, y)
ax = pl.gca()

def pi_formatter(x, pos):
    """
 比较罗嗦地将数值转换为以pi/4为单位的刻度文本
 """
    m = np.round(x / (np.pi/4))
    n = 4
    if m%2==0: m, n = m/2, n/2
    if m%2==0: m, n = m/2, n/2
    if m == 0:
        return "0"
    if m == 1 and n == 1:
        return "$\pi$"
    if n == 1:
        return r"$%d \pi$" % m
    if m == 1:
        return r"$\frac{\pi}{%d}$" % n
    return r"$\frac{%d \pi}{%d}$" % (m,n)

# 设置两个坐标轴的范围
pl.ylim(-1.5,1.5)
pl.xlim(0, np.max(x))

# 设置图的底边距
pl.subplots_adjust(bottom = 0.15)

pl.grid() #开启网格

# 主刻度为pi/4
ax.xaxis.set_major_locator( MultipleLocator(np.pi/4) )

# 主刻度文本用pi_formatter函数计算
ax.xaxis.set_major_formatter( FuncFormatter( pi_formatter ) )

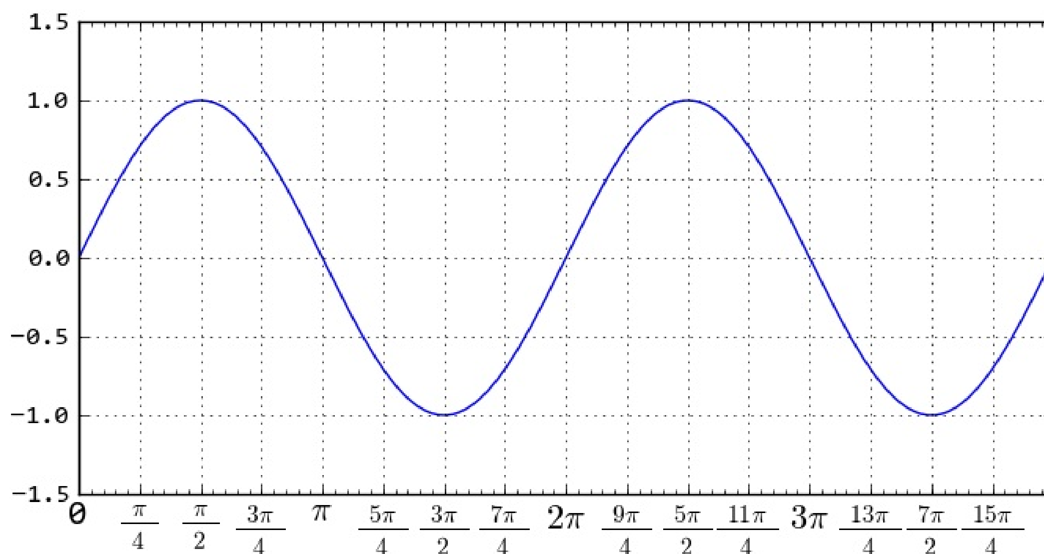
# 副刻度为pi/20
ax.xaxis.set_minor_locator( MultipleLocator(np.pi/20) )

# 设置刻度文本的大小
for tick in ax.xaxis.get_major_ticks():
    tick.label1.set_fontsize(16)
pl.show()
```

关于刻度的定位和文本格式的东西都在matplotlib.ticker中定义，程序中使用到如下两个类：

- **MultipleLocator** : 以指定值的整数倍为刻度放置刻度线
- **FuncFormatter** : 使用指定的函数计算刻度文本，他会传递给所指定的函数两个参数：刻度值和刻度序号，程序中通过比较笨的办法计算出刻度值所对应的刻度文本

此外还有很多预定义的Locator和Formatter类，详细内容请参考相应的API文档。



手工配置X轴的刻度线的位置和文本，并开启副刻度

Traits-为Python添加类型定义

Python作为一种动态编程语言，它的变量没有类型，这种灵活性给快速开发带来很多便利，不过它也不是没有缺点。Traits库的一个很重要的目的就是为了解决这些缺点所带来的问题。

背景

Traits库最初是为了开发Chaco(一个2D绘图库)而设计的，绘图库中有很多绘图用的对象，每个对象都有很多例如线型、颜色、字体之类的属性。为了方便用户使用，每个属性可以允许多种形式的值。例如，颜色属性可以是

- 'red'
- 0xff0000
- (255, 0, 0)

也就是说可以用字符串、整数、组元等类型的值表达颜色，这样的需求初看起来用Python的无类型变量是一个很好的选择，因为我们可以把各种各样的值赋值给颜色属性。但是颜色属性虽然可以接受多样的值，却不是能接受所有的值，比如"abc"、0.5等等就不能很好地表示颜色。而且虽然为了方便用户使用，对外的接口可以接受各种各样形式的值，但是在内部必须有一个统一的表达方式来简化程序的实现。

用Trait属性可以很好地解决这样的问题:

- 它可以接受能表示颜色的各种类型的值
- 当给它赋值为不能表达颜色的值时，它能够立即捕捉到错误，并且提供一个有用的错误报告，告诉用户它能够接受什么样的值
- 它提供一个内部的标准的颜色表达方式

让我们来看一下使用traits属性表示颜色的例子:

```
from enthought.traits.api import HasTraits, Color

class Circle(HasTraits):
    color = Color
```

这个程序从enthought.traits.api中导入我们需要使用的两个对象: HasTraits和Color。所有拥有trait属性的类都需要从HasTraits继承。由于Python的多继承特性，我们很容易将现有的类改为支持trait属性。Color是一个TraitFactory对象，我们在Circle类的定义中用它来声明一个color属性。

熟悉Python的朋友可能会对这个程序觉得有些奇怪：按照标准的Python语法，直接在class下定义的属性color应该是属于Circle类的属性。而我们这里是希望给Circle类的实例一个color属性，是不是应该在初始化函数__init__中运行color = Color呢？

答案是否定的，请记住trait属性像类的属性一样定义，像实例的属性一样使用，我们不管HasTraits是如何实现这一点的，先来看看如何使用trait属性:

```
>>> c = Circle()
>>> Circle.color
Traceback (most recent call last):
AttributeError: type object 'Circle' has no attribute 'color'
>>> c.color
wx.Colour(255, 255, 255, 255)
```

我们看到Circle类没有color属性，而它的实例c则有一个color属性，其缺省值为wx.Colour(255, 255, 255, 255)。

```
>>> c.color = "red"
>>> c.color
wx.Colour(255, 0, 0, 255)
>>> c.color = 0x00ff00
>>> c.color
wx.Colour(0, 255, 0, 255)
>>> c.color = (0, 255, 255)
>>> c.color
wx.Colour(0, 255, 255, 255)
>>> c.color = 0.5
Traceback (most recent call last):
  File "c:\python25\lib\site-packages\Traits-3.1.0-py2.5-win32.egg\
 traits\trait_handlers.py", line 175, in error value )
TraitError: The 'color' trait of a Circle instance must be a string
(r,g,b) or (r,g,b,a) where r, g, b, and a are integers from 0 to 25
instance, an integer which in hex is of the form 0xRRGGBB, where RR
green, and BB is blue or 'aquamarine' or 'black' or 'blue violet' o
'brown' or 'cadet blue' or 'coral' or 'cornflower blue' or 'cyan' o
多英文颜色名... or 'yellow', but a value of 0.5 <type 'float'> was sp
```

c.color支持"red"、0x00ff00和(0, 255, 255)等值。但它不支持0.5这样的浮点数，于是一个很详细的出错信息告诉我们它所有能支持的值。

在开始下一节之前，最后来看一个很酷的东西:

```
>>> c.configure_traits()
True
>>> c.color
wx.Colour(64, 34, 117, 255)
```

执行c.configure_traits()之后，出现如下的对话框以供我们修改颜色属性，任意选择一个颜色、按OK按钮，看到configure_traits函数返回True，而c.color已经变为我们所选择的颜色了。注意你需要在iPython -wthread或者spyder下运行此函数，否则会出现对话框不响应的问题。



自动生成的修改颜色Trait属性的对话框

Traits是什么

trait为Python对象的属性增加了类型定义的功能，此外还提供了如下的额外功能：

- 初始化：每个trait属性都定义有自己的缺省值，这个缺省值用来初始化属性
- 验证：基于trait的属性都有明确的类型定义，只有满足定义的值才能赋值给属性
- 委托：trait属性的值可以委托给其他对象的属性
- 监听：trait属性的值的改变可以触发指定的函数的运行
- 可视化：拥有trait属性的对象可以很方便地提供一个用户界面交互式地改变trait属性的值

下面这个简单的例子展示了trait所提供的这五项能力：

```
from enthought.traits.api import Delegate, HasTraits, Instance, Int

class Parent ( HasTraits ):
    # 初始化: last_name被初始化为'Zhang'
    last_name = Str( 'Zhang' )

class Child ( HasTraits ):
    age = Int

    # 验证: father属性的值必须是Parent类的实例
    father = Instance( Parent )

    # 委托: Child的实例的last_name属性委托给其father属性的last_name
    last_name = Delegate( 'father' )

    # 监听: 当age属性的值被修改时，下面的函数将被运行
    def _age_changed ( self, old, new ):
        print 'Age changed from %s to %s ' % ( old, new )
```

下面用这两个类创建立两个实例:

```
>>> p = Parent()
>>> c = Child()
```

由于没有设置c的father属性，因此无法获得它的last_name属性:

```
>>> c.last_name
Traceback (most recent call last):
AttributeError: 'NoneType' object has no attribute 'last_name'
```

设置father属性之后，我们就可以得到c的last_name了:

```
>>> c.father = p
>>> c.last_name
'Zhang'
```

设置c的age属性将触发_age_changed方法的执行:

```
>>> c.age = 4
Age changed from 0 to 4
```

调用configure_traits:

```
>>> c.configure_traits()
True
```

弹出一个如下的对话框，用户可以通过它修改c的trait属性，



为Child类自动生成的属性修改对话框

可以看到属性按照其英文名排序，垂直排为一列。由于father属性是Parent类的实例，所以它给我们一个按钮，点此按钮出现下面的设置father对象的tratis属性的对话框



点击Child对话框中的Father按钮之后，弹出编辑father属性的对话框

在上面这个对话框中修改father的Last name，可以看到child的Last name属性也随之发生变化。

我们可以调用print_traits方法输出所有的trait属性与其值:

```
>>> c.print_traits()
age:       4
father:    <__main__.Parent object at 0x13B49120>
last_name: u'Zhang'
```

调用get方法获得一个描述对象所有trait属性的dict:

```
>>> c.get()
{'age': 4, 'last_name': u'Zhang', 'father': <__main__.Parent object
```

此外还可以调用set方法设置trait属性的值，set方法可以同时配置多个trait的属性:

```
>>> c.set(age = 6)
Age changed from 4 to 6
<__main__.Child object at 0x13B494B0>
```

动态添加Trait属性

前面介绍的方法都是在类的定义中声明Trait属性，在类的实例中使用Trait属性。由于Python是动态语言，因此Traits库也提供了为某个特定的实例添加Trait属性的方法。

下面的例子，直接产生HasTraits类的一个实例a, 然后调用其add_trait方法动态地为a添加一个名为x的Trait属性，其类型为Float，初始值为3.0。

```
>>> from enthought.traits.api import *
>>> a = HasTraits()
>>> a.add_trait("x", Float(3.0))
>>> a.x
3.0
```

接下来再创建一个HasTraits类的实例b，用add_trait方法为b添加一个属性a，指定其类型为HasTraits类的实例。然后把实例a赋值给实例b的属性a：b.a。

```
>>> b = HasTraits()
>>> b.add_trait("a", Instance(HasTraits))
>>> b.a = a
```

然后为实例b添加一个类型为Delegate(代理)的属性y，它是b的属性a所表示的实例的属性x的代理，即b.y是b.a.x的代理。注意我们在用Delegate声明代理时，第一个参数b的一个属性名"a"，第二个参数是是此属性的属性名"x"，modify=True表示可以通过b.y修改b.a.x的值。我们看到当将b.y的值改为10的时候，a.x的值也同时改变了。

```
>>> b.add_trait("y", Delegate("a", "x", modify=True))
>>> b.y
3.0
>>> b.y = 10
>>> a.x
10.0
```

Property属性

标准的Python提供了Property功能，Property看起来像对象的一个成员变量，但是在获取它的值或者给它赋值的时候实际上是调用了相应的函数。Traits也提供了类似的功能。让我们先来看一个例子：

```
# -*- coding: utf-8 -*-
# filename: traits_property.py
from enthought.traits.api import HasTraits, Float, Property, cachec

class Rectangle(HasTraits):
    width = Float(1.0)
    height = Float(2.0)

    #area是一个属性，当width,height的值变化时，它对应的_get_area函数将被调

    area = Property(depends_on=['width', 'height'])

    # 通过cached_property decorator缓存_get_area函数的输出
    @cached_property
    def _get_area(self):
        """
   area的get函数，注意此函数名和对应的Proerty名的关系
   """
        print 'recalculating'
        return self.width * self.height
```

在Rectangle类定义中，使用Property()定义了一个area属性。Traits所提供的Property和标准Python的有所不同，Traits中根据属性名直接决定了它的访问函数，当用户读取area值时，将得到_get_area函数的返回值；而设置area的值时，_set_area函数将被调用。此外，通过关键字参数depends_on，指定当width和height属性变化时自动计算area属性。

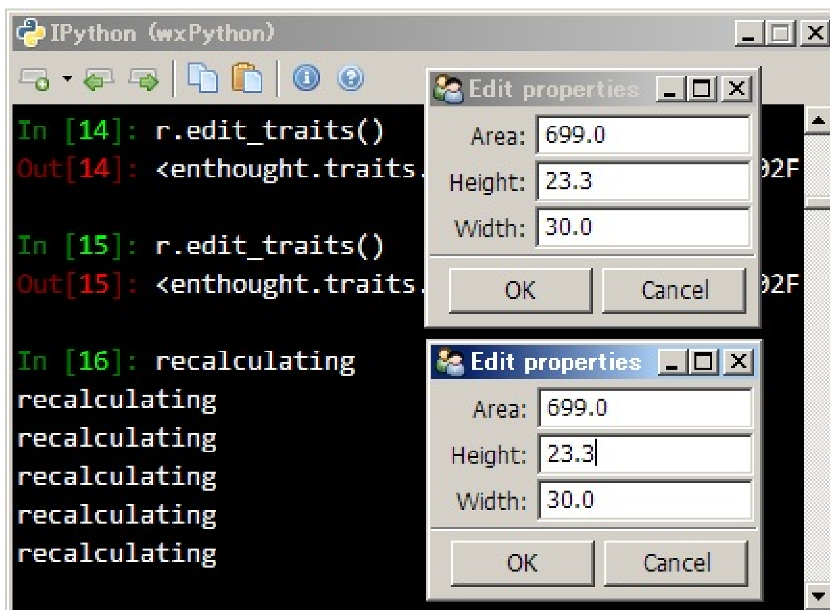
在_get_area函数用@cached_property进行修饰，使得_get_area函数的返回值将被缓存，除非width和height的值发生变化，否则将一直使用缓存的值。下面我们来看看Rectangle的用法。在traits_property.py的文件夹下，启动IPython -wthread：

```
>>> run traits_property.py
>>> r = Rectangle()
>>> r.area  # <-- 第一次取得area，需要进行运算
recalculating
2.0
>>> r.width = 10
>>> r.area # <--修改width之后，取得area，需要进行计算
recalculating
20.0
>>> r.area # <--width和height都没有发生变化，因此直接返回缓存值，没有重新计
20.0
```

我们看到通过depends_on和@cached_property，系统可以跟踪area属性的状态，判断是否需要调用_get_area函数重新计算area的值。注意在运行r.width=10时，并没有立即运行_get_area函数，这是因为系统知道没有任何物体在监听r.area属性，因此它只是保存一个需要重新计算的标志。等到真正需要获取area的值时，再调用_get_area函数。

如果我们调用r.edit_traits()，就会看到depends_on的强大功能了。为了更加有趣一些，这里连续调用两次edit_traits，弹出两个编辑界面：

```
>>> r.edit_traits()
<enthought.traits.ui.ui.UI object at 0x02FCD420>
>>> r.edit_traits()
<enthought.traits.ui.ui.UI object at 0x02FD68A0>
```



修改两个对话框中的任意个Height或者Width属性都会重新计算Area，并同时更新对话框显示

然后修改任何一个界面中的width或者height属性，你可以注意到在输入数值的同时，两个界面中的Area，Height和Width等各个文本框同时更新，每次键盘按键都会调用_get_area函数。此时在IPython窗口修改width的值的话，也会调用_get_area函数：

```
>>> r.width = 25
recalculating
```

当打开界面之后，界面对象开始监听对象r的各个属性，因此当我们修改r.width之后，系统设置r.area的标志为需要重新计算，然后发现r.area的值有对象在监听，因此直接调用_get_area函数更新其值，并且通知所有的监听对象，因此界面就一齐更新了。

让我们来看看在traits的内部，是如何处理属性值的改变引起界面变化的：

```
# -*- coding: utf-8 -*-
# filename: traits_listener.py
from enthought.traits.api import *

class Child ( HasTraits ):
    name = Str
    age = Int
    doing = Str

    def __str__(self):
        return "%s<%x>" % (self.name, id(self))

    # 通知: 当age属性的值被修改时，下面的函数将被运行
    def _age_changed ( self, old, new ):
        print "%s.age changed: form %s to %s" % (self, old, new)

    def _anytrait_changed(self, name, old, new):
        print "anytrait changed: %s.%s from %s to %s" % (self, name

def log_trait_changed(obj, name, old, new):
    print "log: %s.%s changed from %s to %s" % (obj, name, old, new

if __name__ == "__main__":
    h = Child(name = "HaiYue", age=4)
    k = Child(name = "KaiYu", age=1)
    h.on_trait_change(log_trait_changed, name="doing")
```

Child类有一个age属性，当其值发生变化时，其对应的静态监听函数_age_changed 将被调用，而 _anytrait_changed则是一个特殊的静态监听函数，HasTraits对象的任何trait属性值的改变都会调用此函数。

log_trait_changed是一个普通函数。通过h.on_trait_change调用动态地将其与h的doing属性联系起来，即当h对象的doing属性改变时，log_trait_changed函数将被调用。

在IPython中运行上面的程序：

```
>>> run traits_listener.py
anytrait changed: <201ba80>.age from 0 to 4
<201ba80>.age changed: form 0 to 4
anytrait changed: HaiYue<201ba80>.name from  to HaiYue
anytrait changed: <201bae0>.age from 0 to 1
<201bae0>.age changed: form 0 to 1
anytrait changed: KaiYu<201bae0>.name from  to KaiYu
```

然后分别改变h和k这两个对象的各个属性：

```
>>> h.age = 5
anytrait changed: HaiYue<5d87e70>.age from 4 to 5
HaiYue<5d87e70>.age changed: form 4 to 5
>>> h.doing = "sleeping"
anytrait changed: HaiYue<5d87e70>.doing from  to sleeping
log: HaiYue<5d87e70>.doing changed from  to sleeping
>>> k.doing = "playing"
anytrait changed: KaiYu<5d874e0>.doing from  to playing
```



Trait属性的监听函数的调用顺序

静态监听函数的参数有如下几种形式：

- _age_changed(self)
- _age_changed(self, new)
- _age_changed(self, old, new)
- _age_changed(self, name, old, new)

而动态监听函数的参数有如下几种：

- observer()
- ovserver(new)
- ovserver(name, new)
- ovserver(obj, name, new)
- ovserver(obj, name, old, new)

其中obj表示属性发生变化的对象，name为发生改变的属性名，old为改变前的值，new为现在值。

动态监听函数不但可是普通函数，还可以是某个对象的方法。

当多个trait属性都需要同一个静态监听函数时，用固定函数名就比较麻烦了：你需要写多个_xxx_changed函数，其中再调用某个函数进行同样的处理。Trait库提供的解决方案是：用@on_trait_changed对监听函数进行修饰：

TraitsUI-轻松制作用户界面

Python有着丰富的界面开发库，除了缺省安装的Tkinter以外，wxPython、pyQt4等都是非常优秀的界面开发库。但是它们有一个共同的问题：需要开发者掌握众多的API函数，许多细节，例如配置控件的属性、位置以及事件响应都需要开发者一一处理。

在开发科学计算程序时，我们希望快速实现一个够用的界面，让用户能够交互式的处理数据，而又不希望在界面制作上花费过多的精力。以traits为基础、以Model-View-Controller为设计思想的TraitUI库就是实现这一理想的最佳伴侣。

缺省界面

TraitsUI是一套建立在Traits库基础上的用户界面库。它和Traits紧密相连，如果你已经设计好了一个继承于HasTraits的类的话，那么直接调用其configure_traits方法，系统将会使用TraitsUI自动生成一个界面，以供用户交互式地修改对象的trait属性。让我们先来看下面这个例子:

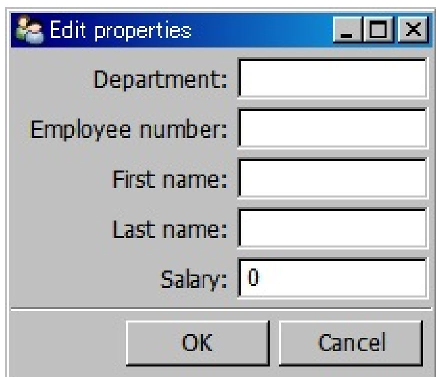
```
from enthought.traits.api import HasTraits, Str, Int

class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

sam = SimpleEmployee()
sam.configure_traits()
```

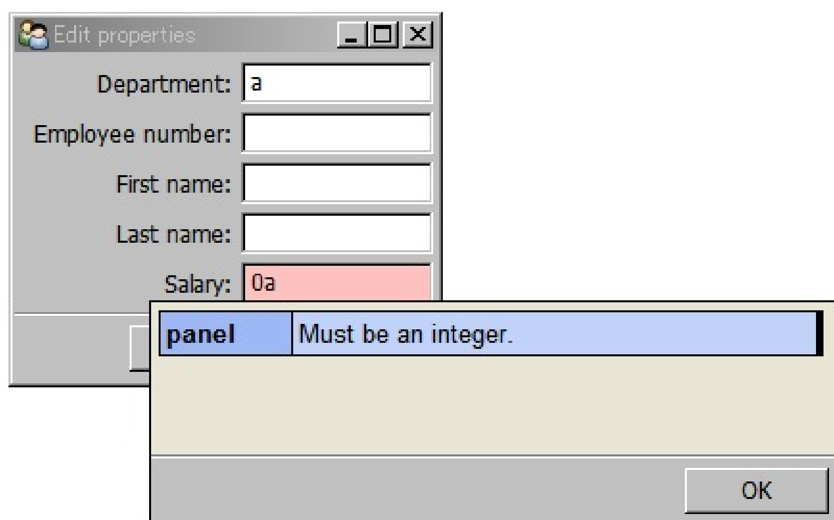
此程序创建一个SimpleEmployee类的对象sam，然后调用sam.configure_traits显示出如下的缺省界面：



自动生成的SimpleEmployee类的对话框

可以看到此界面是自动根据trait属性生成。所有的属性都以文本框的形式编辑，并且每个文本框前面都有一个文字标签，其文字根据trait属性名自动生成：第一个字母变为大写，所有的下划线变为空格。最下面为我们提供了OK和Cancel按钮以确定或者取消对trait属性值的修改。

salary属性虽然和其它属性一样都采用文本框进行编辑，但是由于salary属性定义为Int类型，所以它将检查非法输入，并以红色背景警示，鼠标左击Salary标签，将弹出salary属性相关的详细说明，由于我们没有设置此说明，系统缺省给出salary所能接受的值的类型。



界面中的每个属性编辑器都有详细说明，并且能检查非法输入

我们连一行界面相关的代码都没有写，却能得到这样一个已经够实用的界面，应该还是很令人满意的吧。为了人工控制界面的设计和布局，就需要我们添加自己的代码了。

自定义界面

下面的程序在前面的基础上自定义了一个视图对象view1，然后将此对象传递给configure_traits方法，于是界面就按照视图中描述的那样生成了：

```
# -*- coding: utf-8 -*-
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item

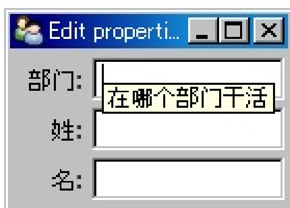
class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str
    employee_number = Str
    salary = Int

view1 = View(
    Item(name = 'department', label=u"部门", tooltip=u"在哪个部门干活'
    Item(name = 'last_name', label=u"姓"),
    Item(name = 'first_name', label=u"名"))

sam = SimpleEmployee()
sam.configure_traits(view=view1)
```

选择后台界面库

用traits.ui库创建的界面可以选择后台界面库，目前支持的有qt4和wx两种。在启动程序时添加 -toolikt qt4 或者 -toolikt wx 选择使用何种界面库生成界面。本文中全部使用wx作为后台界面库。



通过label和tooltip手工指定属性编辑器的标签和说明

有关界面视图的对象都在traits.ui库中，所以首先从其中载入View和Item。View用来生成视图，而Item则用来描述视图中的项目(控件)。程序中，用Item依次创建三个视图项目，都作为参数传递给View，于是所生成的界面中按照参数的顺序显示控件，而不是按照trait属性名排序了。

Item对象

Item对象是视图的基本组成单位，每个Item描述界面中的中的一个控件，通常都是用来显示HasTraits对象中的某一个trait属性。每个Item由一系列的关键字参数来进行配置，这些参数对Item的内容、表现以及行为进行描述。其中最重要的一个参数就是name。我们看到name参数的值都配置为SimpleEmployee类的trait属性名，于是Item就知道到哪里去寻找真正要显示的值了。可以看出视图与数据是通过属性名联系起来的。剩下的两个参数label和tooltip设置Item在界面中的一些显示相关的属

性。Item对象还有很多属性其它属性，请参考TraitsUI的用户手册，或者在iPython中输入Item??直接查看其源代码。如果你查看了Item的源代码的话，你就会发现，原来Item的这些属性也都是用trait定义的：

```
class Item ( ViewSubElement ):
    """ An element in a Traits-based user interface.
 """

    #  Trait definitions:

    # A unique identifier for the item. If not set, it defaults to
    # of **name**.
    id = Str

    # User interface label for the item in the GUI. If this attribu
    # set, the label is the value of **name** with slight modificat
    # underscores are replaced by spaces, and the first letter is c
    # If an item's **name** is not specified, its label is displaye
    # static text, without any editor widget.
    label = Str

    # Name of the trait the item is editing:
    name = Str
```

除了Item之外，TraitsUI库还定义了下面几个Item的子类：

- Label
- Heading
- Spring

这些类用来协助View的布局，因此不需要和某个trait属性关联。

Group对象

前面的例子中，我们通过把三个Item对象传递给View，创建了一个控件垂直排列的布局。然而在真正的界面开发中，需要更加高级的布局方式，例如，将一组相关的元素组织在一起，放到一个组中，我们可以为此组添加标签，定义组的帮助文本，通过设置组的属性使组类的元素同时有效或无效。在TraitUI中，这样的组的功能通过Group对象实现，让我们来修改一下前面的例子：

```
# -*- coding: utf-8 -*-
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item, Group

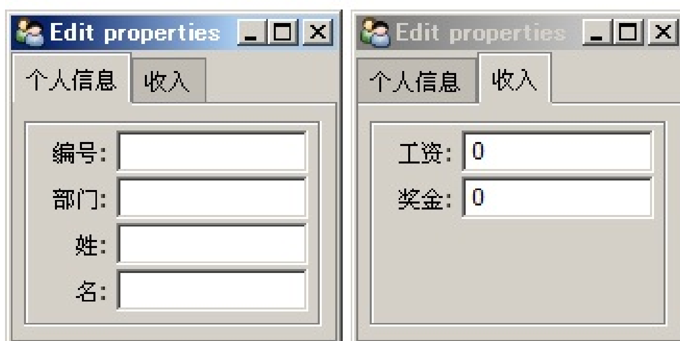
class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int
    bonus = Int

view1 = View(
    Group(
        Item(name = 'employee_number', label=u'编号'),
        Item(name = 'department', label=u"部门", tooltip=u"在哪个部门
        Item(name = 'last_name', label=u"姓"),
        Item(name = 'first_name', label=u"名"),
        label = u'个人信息',
        show_border = True
    ),
    Group(
        Item(name = 'salary', label=u"工资"),
        Item(name = 'bonus', label=u"奖金"),
        label = u'收入',
        show_border = True
    )
)

sam = SimpleEmployee()
sam.configure_traits(view=view1)
```

此程序的运行效果如下：



分标签页显示两个Group的内容

我们分别创建两个Group传递给View，每个Group中仍然通过Item创建控件，通过Group的关键字参数指定其label和show_border属性。由于View中的所有内容都是Group，它自动地将两个Group放到Tab中，对两个Group进行分标签显示。

如果我们希望能同时看到两个Group的话，可以另外再创建一个Group将这两个Group包括起来：

```
view2 = View( Group( view1.content ) )
```

这里我们创建视图view2，它包括一个Group，此Group的内容则直接使用view1的内容(也就是那两个Group)。当然也可以把view1中的内容复制进去：

```
view2 = View(Group(
    Group(
        Item(name = 'employee_number', label=u'编号'),
        Item(name = 'department', label=u"部门", tooltip=u"在哪个部门
        Item(name = 'last_name', label=u"姓"),
        Item(name = 'first_name', label=u"名"),
        label = u'个人信息',
        show_border = True
        ),
    Group(
        Item(name = 'salary', label=u"工资"),
        Item(name = 'bonus', label=u"奖金"),
        label = u'收入',
        show_border = True
        )
))
```

然后我们将view2传递给configure_traits，用view2显示界面：

```
sam.configure_traits(view=view2)
```



竖排显示两个Group的内容

在创建Group时，我们可以通过设置其orientation和layout等属性，改变Group的内容呈现方式。由于某些设置会经常用到，因此还提供了专门的Group子类重载这些属性的缺省值。例如下面是从Group类继承的HSplit类的代码：

```
class HSplit ( Group ):
    # ... ...
    layout      = 'split'
    orientation = 'horizontal'
```

HSplit对象将其所包括的内容按照水平排列，并且在每两个子内容之间添加一个可调整的分隔条，HSplit和如下的代码是等价的：

```
Group( ... , layout = 'split', orientation = 'horizontal')
```

为了正确显示分隔条，其子内容中需要有一个具有scrollable属性，如下面的代码(省略Item定义等部分)所示：

```
Group(orientation= 'horizontal')
```

- **HFlow**：内容水平排列，当超过水平宽度时，将自动换行：

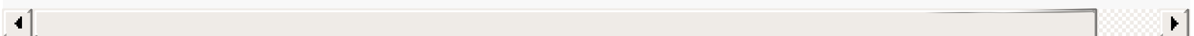
  ```
  Group(orientation= 'horizontal', layout='split')
  ```

  - **Tabbed**：内容分标签页显示：

  ```
  Group(orientation= 'vertical')
  ```

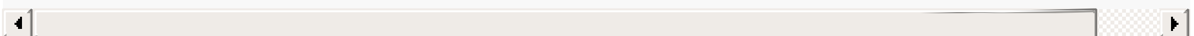
  - **VFlow**：内容垂直排列，当超过垂直高度时，将自动换列：

  ```
  Group(orientation= 'vertical', layout='flow', show_labels=False
  ```

  - **VFold**：内容垂直排列，可折叠 ：

  ```
  Group(orientation= 'vertical', layout='fold', show_labels=False
  ```

  - **VGrid**：按照两列的网格进行垂直排列 ：

  ```
  Group(orientation= 'vertical', columns=2)
  ```

  - **VSplit** ： 内容垂直排列，中间插入分隔条：

  ```
  Group(orientation= 'vertical', layout='split')
  ```

配置视图

前面介绍了如何使用Item和Group等类组织窗口界面中的内容，这一节我们来看看如何配置窗口本身的属性。

视图类型

通过kind属性可以修改View对象的显示类型：

- 'modal' : 模式窗口, 非即时更新
- 'live' : 非模式窗口，即时更新
- 'livemodal' : 模式窗口，即时更新
- 'nonmodal' : 非模式窗口，非即时更新
- 'wizard' : 向导类型
- 'panel' : 嵌入到其它窗口中的面板，即时更新，非模式
- 'subpanel'

其中 'modal', 'live', 'livemodal', 'nonmodal' 四种类型的View都将采用窗口显示其内容。所谓模式窗口，表示此窗口关闭之前，程序中的其它窗口都不能被激活。而即时更新则是指当窗口中的控件内容改变时，修改会立即反应到窗口所对应的模型数据上，非即时更新的窗口则会复制模型数据，所有的改变在模型副本上进行，只有当用户确定修改(通常通过OK或者Apply按钮)时，才会修改原始数据。

'wizard'由一系列特定的向导窗口组成，属于模式窗口，并且即时更新数据。

'panel'和'subpanel' 则是嵌入到窗口中的面板，panel可以拥有自己的命令按钮，而subpanel则没有命令按钮。

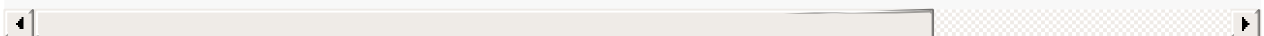
命令按钮

在对话框中经常可以看到 OK, Cacel, Apply 之类的按钮，我们称之为命令按钮，它们完成所有对话框窗口都共同的操作。在TraitsUI中，这些按钮可以通过View对象的buttons属性进行设置，其值为要显示的按钮列表。

TraitsUI定义了UndoButton, ApplyButton, RevertButton, OKButton, CancelButton等六个标准的命令按钮，每个按钮对应一个名字，在指定buttons属性时，可以使用按钮的类名或者其对应的名字。与按钮类对应的名字就是类名除去Button，例如UndoButton对应为"Undo"。

在 enthought.tratis.ui.menu 中还预定义了一些命令按钮列表，方便直接使用：

```
OKCancelButtons = ``[OKButton, CancelButton ]``
ModalButtons = ``[ ApplyButton, RevertButton, OKButton, CancelButto
LiveButtons = ``[ UndoButton, RevertButton, OKButton, CancelButton,
```



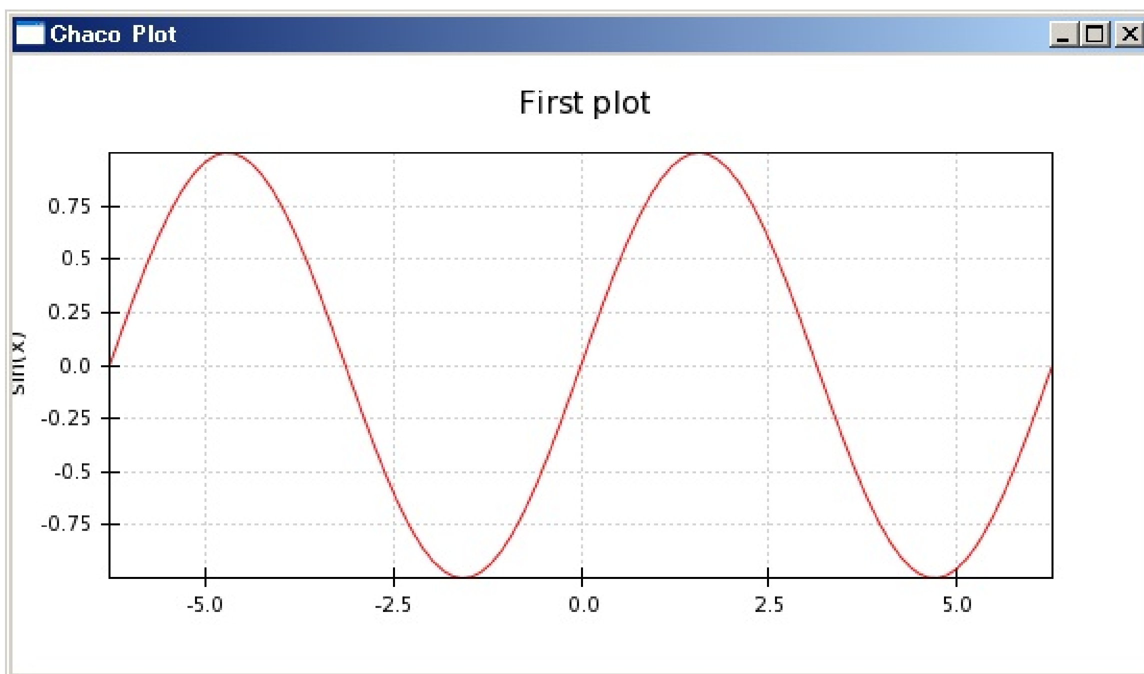
Chaco-交互式图表

Chaco是一个2D的绘图库，如果你安装了Python(x,y)的话，可以在pythonxy的安装目录下的找到Chaco的demo程序:

```
import numpy as np
from enthought.chaco.shell import *

x = np.linspace(-2*np.pi, 2*np.pi, 100)
y = np.sin(x)

plot(x, y, "r-")
title("First plot")
ytitle("sin(x)")
show()
```



用Chaco的脚本绘图方式快速绘制正弦波

plot函数的第三个参数中的"r"指定绘图的颜色为红色，"-"指定绘图的线型为实线。title函数为绘图添加标题，ytitle为Y轴添加标题，show()函数最终显示绘图结果。

脚本绘图不是Chaco的强项，虽然它的这套脚本绘图API和Matplotlib的pylab类似，不过它提供的功能却没有pylab丰富。Chaco的优势在于它可以很方便地嵌入到你的应用程序之中，开发出自己独特的绘图应用。

面向应用绘图

要将Chaco嵌入到别的应用程序之中，需要做一些额外的工作，因此代码量比面向脚本绘图要多，不过同时也更具有灵活性。先来看一个例子:

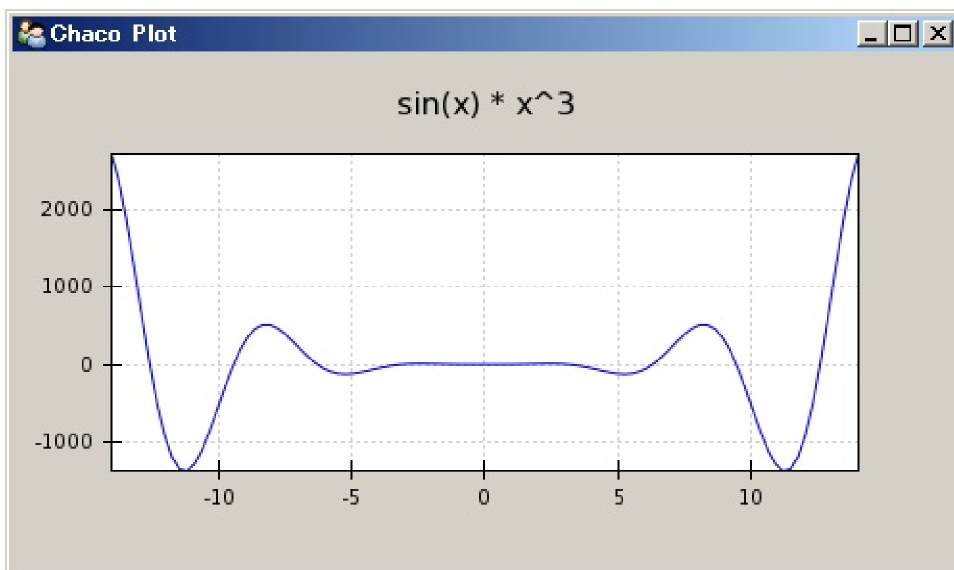
```
from enthought.traits.api import HasTraits, Instance
from enthought.traits.ui.api import View, Item
from enthought.chaco.api import Plot, ArrayPlotData
from enthought.enable.component_editor import ComponentEditor
from numpy import linspace, sin

class LinePlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500, resizable=True, title="Chaco Plot")

    def __init__(self):
        super(LinePlot, self).__init__()
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x=x, y=y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="line", color="blue")
        plot.title = "sin(x) * x^3"
        self.plot = plot

if __name__ == "__main__":
    LinePlot().configure_traits()
```

上面这段代码绘制如下的曲线:



用Chaco的面向对象的方式绘制曲线

这段代码看起来似乎挺复杂，其实只要掌握了其基本设计思想，就很容易理解了。首先是许多import语句，为了保持应用程序的名字空间整洁以及让自动语法检查工具能帮助我们检查代码，这些语句只import了需要的对象。

- HasTraits, Instance：这两个从traits库中导入，HasTraits是所有拥有Trait属性的类的父类，我们自己定义的LinePlot类继承于它。而Instance用来创建一个Trait属性，此属性的值为某个指定的类的实例。
- View, Item：从traits.ui库导入，View用来创建一个生成用户界面用的视图，而Item则用来定义视图中的元素。
- Plot, ArrayPlotData：从chaco库中导入，Plot本身是一个描述绘图的类，它的祖先类中有HasTraits，因此它本身也是一个拥有trait属性的类，ArrayPlotData是用来统一保存绘图所用的数据的类。也就是说Plot管理绘图，而ArrayPlotData则用来管理绘图所用的数据。
- ComponentEditor：从enable库中导入，用户界面视图中使用ComponentEditor来显示LinePlot类的plot属性。如果trait属性为Int、Str或者Float之类的简单类型的话，系统能够自动的帮我们选择对应的GUI元素来显示它们。但是系统不知道如何显示Chaco中定义的Plot这样的的类型，因此我们必须手工指定采用ComponentEditor来显示Plot。
- linspace, sin：从numpy库中导入，linspace用来产生一个等差数列，numpy中的sin函数可以自动对数组中的每个元素进行计算。

接下来的代码部分:

```
class LinePlot(HasTraits):
    plot = Instance(Plot)
```

首先定义一个LinePlot继承于HasTraits，并且它有一个trait属性plot为Plot类的实例。然后定义视图:

```
traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500, resizable=True, title="Chaco Plot")
```

此视图在LinePlot类中定义，因此在调用configure_traits的时候就不需要指定视图了。视图中有一个元素，它将用来显示名为plot的属性的内容，视图中的元素用Item创建。注意这里使用字符串指定视图元素所对应的属性。然后通过关键字参数editor指定此视图元素采用ComponentEditor进行显示。并且不显示其标签(show_label=False)。通过View的关键字参数width、height、resizable和title分别指定界面的宽、高、是否可改变大小以及其窗口标题栏的文字。

接下来看构造函数，真正的计算在这里:

```
def __init__(self):
    super(LinePlot, self).__init__()
    x = linspace(-14, 14, 100)
    y = sin(x) * x**3
    plotdata = ArrayPlotData(x=x, y=y)
```

在构造函数做其它事情之前，一定要记住调用父类的构造函数，这样HasTraits的功能才能真正在我们的实例中出现。

接下来和脚本绘图一样，计算出绘图所需的x,y坐标的数值数组。然后将这两个数组存到一个ArrayPlotData对象中。ArrayPlotData和字典(dict)有些类似，它将一个字符串(数组的名字)和数组本身联系起来。而真正的绘图对象plot将通过数组的名字在ArrayPlotData中获得数组的内容。这样做就在数据和绘图对象中形成了一个接口界面，修改ArrayPlotData中的数组的值将会立即反应到与此数据相连的绘图对象，而多个绘图对象可以共用ArrayPlotData中的同一数组。

接下来创建绘图对象plot，并且将我们创建的ArrayPlotData实例传递给它，此后plot将在此实例中获取自己绘图所需的数据。:

```
plot = Plot(plotdata)
```

Plot类将Chaco中提供的许多真正用来绘图的对象进行包装，提供了一个统一的接口用来创建和管理这些绘图对象。在今后深入学习Chaco的过程中我们将通过分析Plot类的实现来了解Chaco库的设计思想。

接下来调用plot方法在Plot内部创建真正的绘图对象(一个曲线图):

```
plot.plot(("x", "y"), type="line", color="blue")
```

注意我们传递给plot方法的是数组的名字而不是数组本身，Plot对象会自动通过数组的名字在ArrayPlotData的实例中找到其对应的数组。

然后设置绘图的标题，并且把绘图实例赋值给plot属性:

```
plot.title = "sin(x) * x^3"
self.plot = plot
```

最后是LinePlot对象的实例化和调用configure_trait显示绘图窗口:

```
if __name__ == "__main__":
    LinePlot().configure_traits()
```

由于没有给configure_traits传递视图参数，它将在LinePlot实例中寻找视图的定义，于是它找到traits_view，并且用此视图来显示LinePlot实例的trait属性。于是plot属性将如traits_view中定义的一样，用ComponentEditor显示。

采用和LinePlot类同样的模式，我们可以绘制更多的曲线图：

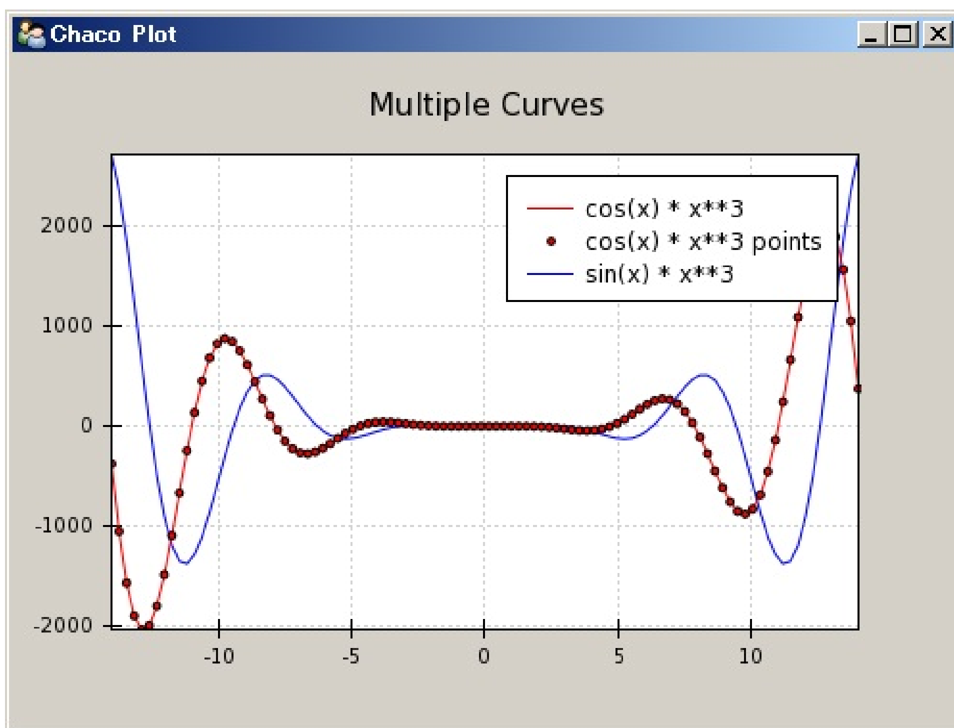
```
from enthought.traits.api import HasTraits, Instance
from enthought.traits.ui.api import View, Item
from enthought.chaco.api import Plot, ArrayPlotData, Legend
from enthought.enable.component_editor import ComponentEditor
from numpy import linspace, sin, cos

class LinePlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500, resizable=True, title="Chaco Plot")

    def __init__(self):
        super(LinePlot, self).__init__()
        x = linspace(-14, 14, 100)
        y1 = sin(x) * x**3
        y2 = cos(x) * x**3
        plotdata = ArrayPlotData(x=x, y1=y1, y2=y2)
        plot = Plot(plotdata)
        plot.plot(("x", "y1"), type="line", color="blue", name="si
        plot.plot(("x", "y2"), type="line", color="red", name="cos
        plot.plot(("x", "y2"), type="scatter", color="red", marker
                  marker_size = 2, name="cos(x) * x**3 points")
        plot.title = "Multiple Curves"
        self.plot = plot

        legend = Legend(padding=10, align="ur")
        legend.plots = plot.plots
        plot.overlays.append(legend)

if __name__ == "__main__":
    lineplot = LinePlot()
    lineplot.configure_traits()
```



绘制多条曲线并且添加图示

在这个程序中，我们调用了3次plot.plot方法，其中两次的type="line"绘制曲线，一次是type="scatter"绘制坐标点。绘制坐标点时通过marker和marker_size配置点的形状和大小。并且我们为每个plot传递了一个name参数。

为了绘制图示(legend)，从chaco.api中载入Legend之后，使用如下三行代码为坐标图添加图示：:

```
legend = Legend(component=plot, padding=10, align="ur")
legend.plots = plot.plots
plot.overlays.append(legend)
```

其中第一行创建一个Legend对象，并且设置padding和align两个属性，padding设置其内容与边框之间的距离，align设置其在容器中的位置: upper right。

为了让legend对象知道要显示什么曲线的图示，我们需要把曲线对象传递给它。三次调用plot绘制的曲线可以通过plot对象plots属性得到，在iPython中运行完上面的程序之后，输入lineplot.plot.plots查看plots属性的值:

```
>>> lineplot.plot.plots
{'cos(x) * x**3 points': [<enthought.chaco.scatterplot.ScatterPlot
'cos(x) * x**3': [<enthought.chaco.lineplot.LinePlot object at 0x14
'sin(x) * x**3': [<enthought.chaco.lineplot.LinePlot object at 0x14
```

我们将plot.plots传递给legend.plots，于是legend就知道要显示哪些曲线的图示了。

最后我们将legend对象添加到plot.overlays中。整个绘图区域分为许多层，每一层放置不同的绘图元素，overlays是最上面的一层，其中放置在屏幕坐标系中的绘图元素。plot的overlays属性为一个TraitListObject类的对象，它继承于list类，具有list的所有能力，因此可以用append方法将绘图元素添加进overlays层。

容器(Container)概述

在Chaco的实现中，Plot类继承于DataView类，而DataView类继承于OverlayPlotContainer类，因此Plot本身就是一个容器。可以把OverlayPlotContainer想象成多张透明绘图纸，我们在多张纸上绘图，然后通过OverlayPlotContainer容器将这些纸张重叠起来，就组成了最终所绘制的图。

除了OverlayPlotContainer容器之外，Chaco还提供了下面几种容器：

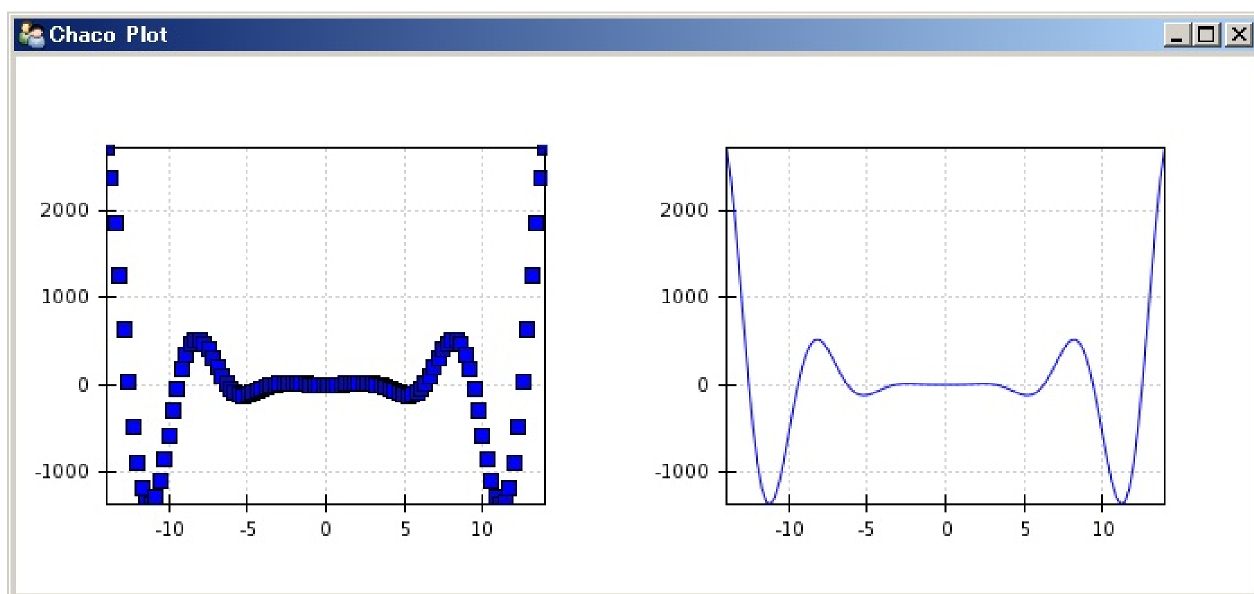
- HPlotContainer：内容横向排列的容器
- VPlotContainer：内容竖向排列的容器
- GridPlotContainer：内容按照网格排列的容器

下面让我们来看一个用HPlotContainer的例子：

```
from enthought.traits.api import HasTraits, Instance
from enthought.traits.ui.api import View, Item
from enthought.chaco.api import HPlotContainer, ArrayPlotData, Plot
from enthought.enable.component_editor import ComponentEditor
from numpy import linspace, sin

class ContainerExample(HasTraits):
    plot = Instance(HPlotContainer)
    traits_view = View(Item('plot', editor=ComponentEditor(), show_
                       width=1000, height=600, resizable=True, titl
    def __init__(self):
        super(ContainerExample, self).__init__()
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x=x, y=y)
        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")
        container = HPlotContainer(scatter, line)
        self.plot = container

if __name__ == "__main__":
    ContainerExample().configure_traits()
```



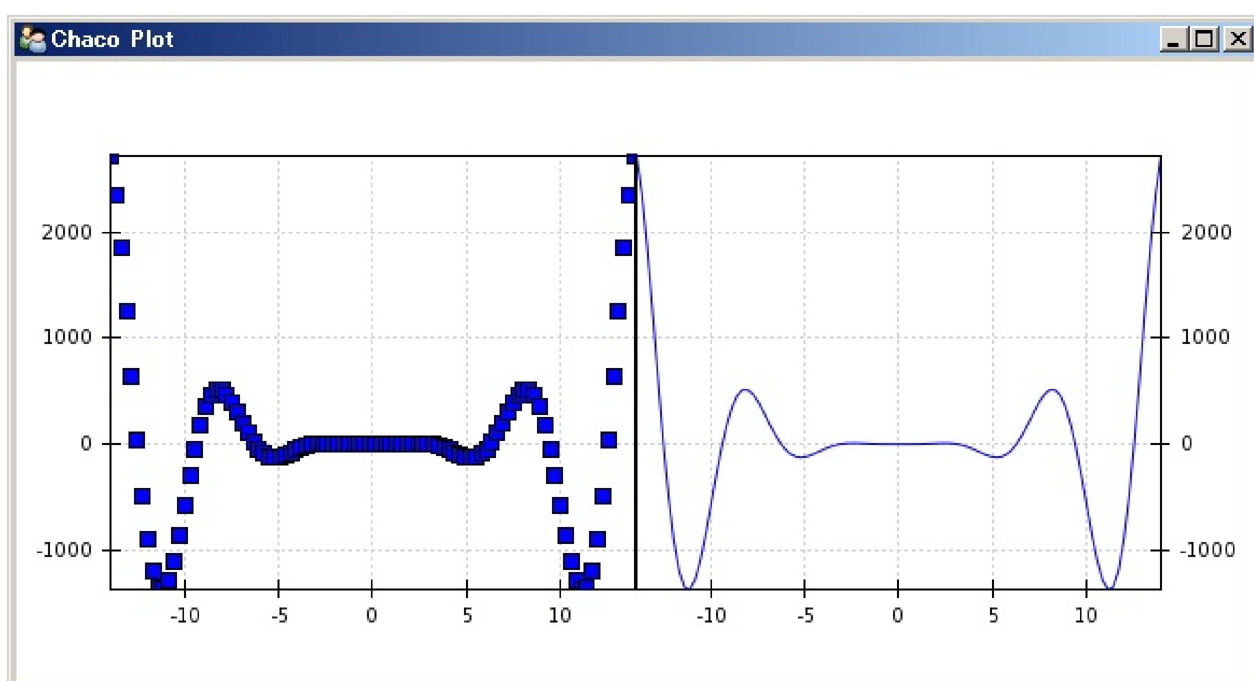
用容器绘制两个子图

这个程序和前面的例子类似，所不同的是：ContainerExample的plot属性不是Plot的实例，而是改为HPlotContainer的实例。这样它就可以横向排列多个图了。

在__init__函数中，用创建两个Plot对象scatter和line，然后创建HPlotContainer对象container，并把两个plot对象传递给它，这样container就知道要水平排列哪些图了。

每个容器都有很多属性可以设置，如果我们在__init__函数最后添加如下几行程序：:

```
scatter.padding_right = 0
line.padding_left = 0
line.y_axis.orientation = "right"
```



修改两个容器的左右padding值，使它们紧靠在一起

我们看到两个Plot之间的间距没有了，他是通过设置容器左图(scatter)的右边距为0，右图(line)的左边距为0来实现的。 并且将右图的Y轴坐标设置到了右边。

编辑绘图属性

到目前为止所绘制的图都是静态的，一旦创建出来就没有办法改变其各种显示属性了。Chaco库是建立在Traits库基础之上的，我们看到的各种各样的对象的属性都是trait属性，这样我们可以使用Traits和TraitsUI的强大功能设置对象的各种属性，下面是一个完整的例子：

```
from enthought.traits.api import HasTraits, Instance, Int, Color
from enthought.traits.ui.api import View, Group, Item
from enthought.enable.component_editor import ComponentEditor
from enthought.chaco.api import marker_trait, Plot, ArrayPlotData
from numpy import linspace, sin

class ScatterPlotTraits(HasTraits):

    plot = Instance(Plot)
    color = Color("blue")
    marker = marker_trait
    marker_size = Int(4)

    traits_view = View(
        Group(Item('color', label="Color"),
              Item('marker', label="Marker"),
              Item('marker_size', label="Size"),
              Item('plot', editor=ComponentEditor(), show_label=Fa
                  orientation = "vertical"),
              width=800, height=600, resizable=True, title="Chaco P

    def __init__(self):
        super(ScatterPlotTraits, self).__init__()
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)

        self.renderer = plot.plot(("x", "y"), type="scatter", colo
        self.plot = plot

    def _color_changed(self):
        self.renderer.color = self.color

    def _marker_changed(self):
        self.renderer.marker = self.marker

    def _marker_size_changed(self):
        self.renderer.marker_size = self.marker_size

if __name__ == "__main__":
    ScatterPlotTraits().configure_traits()
```

为了观察trait控件是如何动态地修改绘图的的各个属性，我用flash录制下对控件的操作，请点击下图下方的播放按钮观看动画。



通过观察上面的这个动画，我们发现对颜色、点型和点的大小等属性的修改立即响应到绘图的属性上。下面我们来分析一下这个程序：

ScatterPlotTraits类定义了4个trait属性，其中一个是我们已经熟知的plot属性，其余的三个分别为color，marker和marker_size。color是一个Color属性，marker_size是一个Int属性，marker比较特别，它是在Chaco的scatter_makers.py中定义的一个Trait属性，采用字典创建，将一个描述点型的字符串映射到点型对应的类，这样我们通过界面上的下拉选择框选择某个点型名称时，在程序内部实际上选择的是其对应的类。

接下来第14行在ScatterPlotTraits类内部定义了一个视图对象traits_view。它创建4个Item分别与4个trait属性对应。为了响应trait属性值的改变事件，我们为类添加了3个事件处理函数_color_changed，_marker_changed和_marker_size_changed。这个三个处理函数通过其名字和trait属性对应，即名为foo的trait属性的缺省事件处理函数名为_foo_changed。值得注意的是这三个处理函数是和trait属性相对应的，而不是界面上的控件。当用户更改了控件的内容之后，此更改自动反映为trait属性值的更改，当trait属性的值更改时，事件处理函数将被运行。这样，当处理函数运行的时候，trait属性的值已经是最新的界面上所显示的值了。因此只需要将此值赋值给曲线绘图对象(render)的对应的属性即可。

那么render对象是什么？我们看到它是plot.plot函数的返回值，这个返回值就是图中所画的那条曲线。前面我们演示了通过plot.plots获得过plot中所有的绘图对象。因此不用render保存此返回值，也可以用plot.plots.values()[0]，或者plot.plots["plot0"]来获取这个绘图对象。"plot0"是系统自动为我们的曲线所起的名字。

TVTK-三维可视化数据

VTK (http://www.vtk.org/) 是一套三维的数据可视化工具，它由C++编写，包涵了近千个类帮助我们处理和显示数据。它在Python下有标准的绑定，不过其API和C++相同，不能体现出Python作为动态语言的优势。因此enthought.com开发了一套TVTK库对标准的VTK库进行包装，提供了Python风格的API、支持Trait属性和numpy的多维数组。本文将以TVTK为标准对VTK的一些功能进行介绍，如果读者已经对VTK很了解，想知道TVTK和VTK的区别的话，可以直接跳到第二节。

TVTK使用简介

a

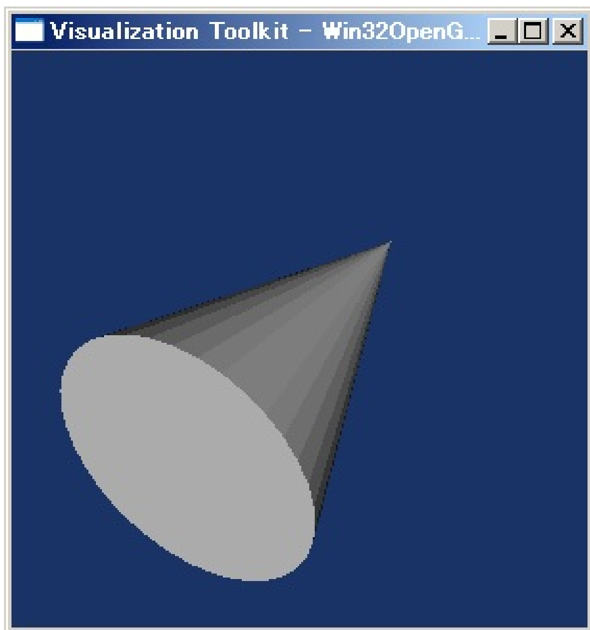
显示圆锥

作为第一例子，让我们来看一个显示圆锥的小程序:

```
# -*- coding: utf-8 -*-
from enthought.tvtk.api import tvtk

# 创建一个圆锥数据源，并且同时设置其高度，底面半径和底面圆的分辨率(用36边形近似
cs = tvtk.ConeSource(height=3.0, radius=1.0, resolution=36)
# 使用PolyDataMapper将数据转换为图形数据
m = tvtk.PolyDataMapper(input = cs.output)
# 创建一个Actor
a = tvtk.Actor(mapper=m)
# 创建一个Renderer，将Actor添加进去
ren = tvtk.Renderer(background=(0.1, 0.2, 0.4))
ren.add_actor(a)
# 创建一个RenderWindow(窗口)，将Renderer添加进去
rw = tvtk.RenderWindow(size=(300,300))
rw.add_renderer(ren)
# 创建一个RenderWindowInteractor (窗口的交互工具)
rwi = tvtk.RenderWindowInteractor(render_window=rw)
# 开启交互
rwi.initialize()
rwi.start()
```

此程序的运行画面如下：



使用TVTK绘制简单的圆锥

首先从tvtk.api中载入tvtk，tvtk像是一个工厂，能够帮助我们创建vtk中的各种对象:

```
>>> from enthought.tvtk.api import tvtk
```

下面创建了一个ConeSource（圆锥数据源）对象，并用变量cs保存它。原始的VTK对象的属性，在tvtk中都以trait属性的形式进行包装，因此我们可以在创建对象的同时，传递关键字参数直接配置各个trait属性的值，在这个例子中，同时设置了圆锥的高度，底面半径和底面圆的分辨率(用36边形近似)等属性，最后调用print_traits显示所创建的圆锥数据的所有trait属性，为了节省篇幅，这里只挑选了其中的几个属性:

```
>>> cs = tvtk.ConeSource(height=3.0, radius=1.0, resolution=36)
>>> cs.print_traits()
...
angle:                              18.43494882292201
...
center:                             array([ 0.,   0.,   0.])
class_name:                         'vtkConeSource'
...
direction:                          array([ 1.,   0.,   0.])
...
height:                             3.0
...
radius:                             1.0
...
resolution:                         36
...
```

在VTK中将原始数据转换为我们看到的屏幕上的一幅图像，要经过许多步骤的处理，这些步骤由众多的VTK的对象共同协调完成，就好象生产线上加工零件一样，每位工人都负责一部分的工作，整条生产线就能将原材料制作成产品。因此在VTK中，这种对象之间协调完成工作的过程被称作流水线(Pipeline)。

原始数据被转换为图像要经过两条流水线：

- 可视化流水线(Visualization Pipeline)：它的工作是将原始数据加工成图形数据。通常我们需要可视化的数据本身并不是图形数据，例如某个零件内部各个部分的温度，或者是流体在各个坐标点上的速度等等。
- 图形流水线(Graphics Pipeline)：它的工作是将将图形数据加工为我们所看到的图像。可视化流水线所产生的图形数据通常是三维空间的数据，如何在二维的屏幕上显示出来就需要图形流水线的加工了。

映射器(Mapper)则是可视化流水线的终点，图形流水线的起点，它将各种派生类能将众多的数据映射为图形数据以供图形流水线加工。

让我们对照一下前面的的圆锥的例子：ConeSource的对象通过程序内部计算输出一组描述圆锥的数据(PolyData)：然后，PolyData通过PolyDataMapper映射器将数据映射为图形数据。在这个例子中，可视化流水线由ConSource和PolyDataMapper组成。

图形数据依次通过Actor、Renderer最终在RenderWindow中显示出来，这一部分就是图形流水线。

- **Actor** : 表示润色场景中的一个实体。它包括一个图形数据(mapper)，并且具有描述实体的位置、方向、大小的属性。
- **Renderer** : 表示润色的场景。它包括多个需要润色的Actor。在圆锥的例子中，它只包括一个表示圆锥的Actor。
- **RenderWindow** : 表示润色用的图形窗口，它包括一个或者多个Render。在圆锥的例子中，它只包括一个Renderer。
- **RenderWindowInteractor** : 给图形窗口提供一些用户交互功能，例如平移、旋转、放大缩小。这些交互式操作并不改变Actor或者图形数据的属性，只是调整场景中的照相机(Camera)的一些设置而已。

什么是PolyData

PolyData是一个描述一组三维空间中的点、线、面的数据结构。点、线、面通过以下几个属性描述：

> - **points** : 类型为Points，保存三维空间中的点的坐标的数组，这些数据不是用来显示的。
> - **verts** : 类型为CellArray，它描述需要显示的顶点，其值为 **points** 某个坐标点的下标，即通过 **verts** 属性描述 **points** 中的哪些点是最终需要显示的。
> - **line** : 类型为CellArray，它描述需要显示的边线，其值为边线的两个端点在 **points** 中的下标。
> - **polys** : 类型为CellArray，它描述需要显示的面，其值为构成面的各个点在 **points** 中的下标。

用ivtk观察流水线

为了方便我们操作和观察流水线，交互式地修改各个tvtk对象的属性，TVTK库为我们提供了一个叫做ivtk的对象。下面是使用ivtk显示圆锥的程序：

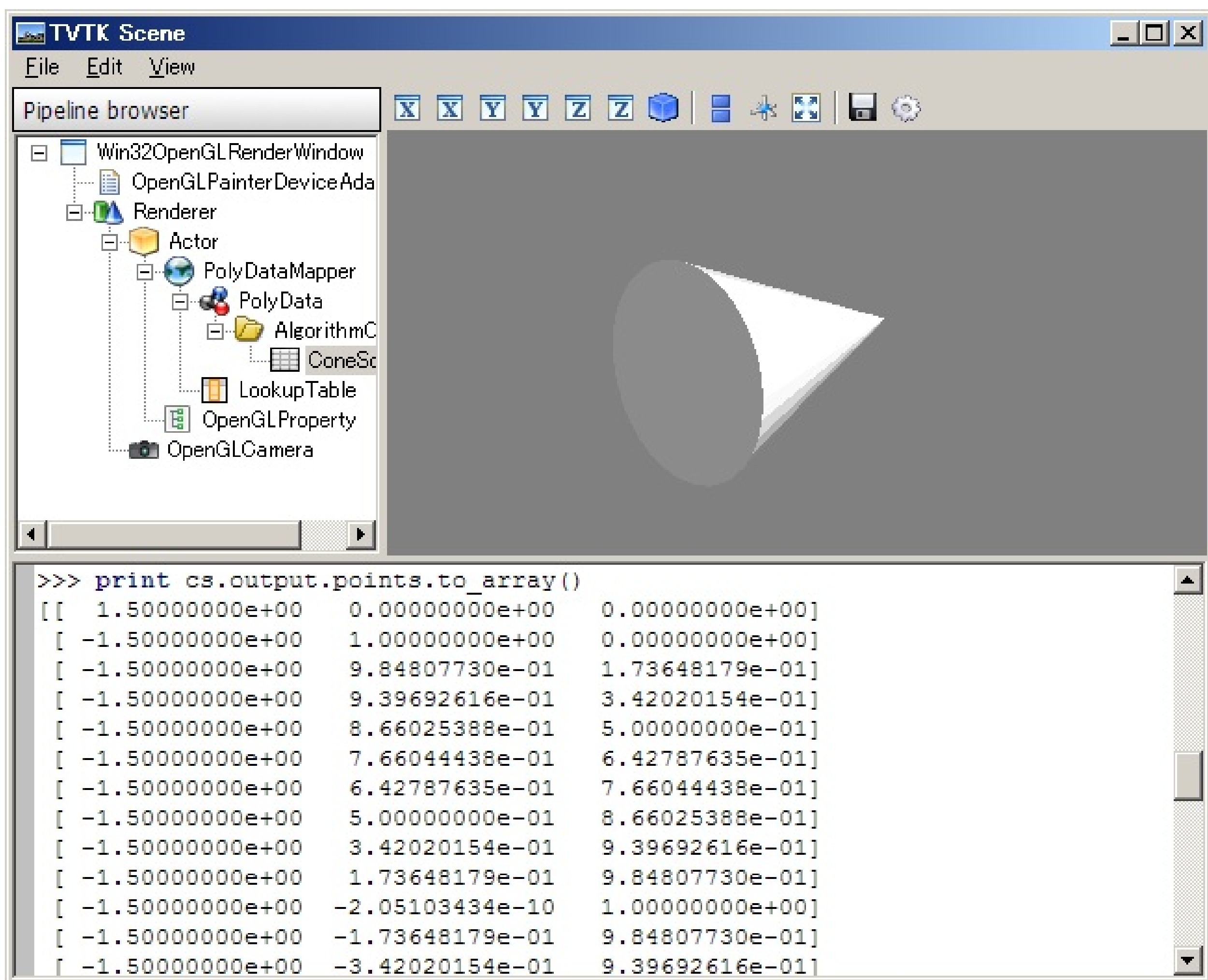
```
# -*- coding: utf-8 -*-
from enthought.tvtk.api import tvtk

# 载入ivtk所需要的对象
from enthought.tvtk.tools import ivtk
from enthought.pyface.api import GUI

cs = tvtk.ConeSource(height=3.0, radius=1.0, resolution=36)
m = tvtk.PolyDataMapper(input = cs.output)
a = tvtk.Actor(mapper=m)

# 创建一个GUI对象，和一个带Crust(Python shell)的ivtk窗口
gui = GUI()
window = ivtk.IVTKWithCrustAndBrowser(size=(800,600))
window.open()
window.scene.add_actor( a ) # 将圆锥的actor添加进窗口的场景中
gui.start_event_loop()
#window.scene.reset_zoom()
```

此程序的运行画面如下：



带流水线浏览器和Python Shell的界面

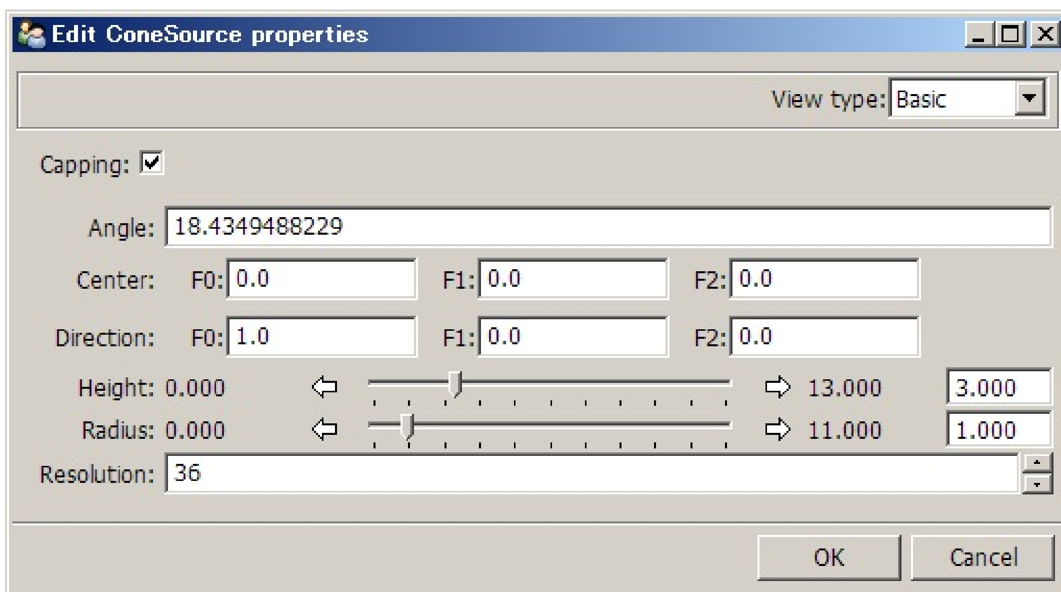
除了显示圆锥的场景之外，ivtk创建的窗口还为我们提供了如下几个元素：

- 场景工具条：位于润色场景的上方。主要提供了各种视角、全屏显示、保存图像等几个功能。
- 流水线浏览器：场景的左边是一个用树状结构表示的流水线。从叶子节点(ConeSource)开始逐步向上层直到根节点RenderWindow，是完整的显示圆锥的流水线。
- **Python Shell**：下方提供了一个Python Shell，便于我们直接输入命令操作各个对象。例如图中我们打印出ConeSource的输出PolyData的points属性的值。即构成圆锥图形的各个点的三维坐标。

流水线浏览器中显示的各个对象的类型都继承于HasTraits类，因此它们都可以提供一个用户界面交互式地修改它们的trait属性。下图是双击ConeSource之后出现的修改ConeSource属性的界面。

Note

在我的电脑上，双击ConeSource之后出现一个很小的窗口，需要手工调整大小。



编辑ConeSource对象的属性的对话框

我们看到可以通过此界面直接修改height、Radius、resolution等属性，并且修改之后场景中的圆锥按照最新的属性值立即更新显示。

从文件读取数据

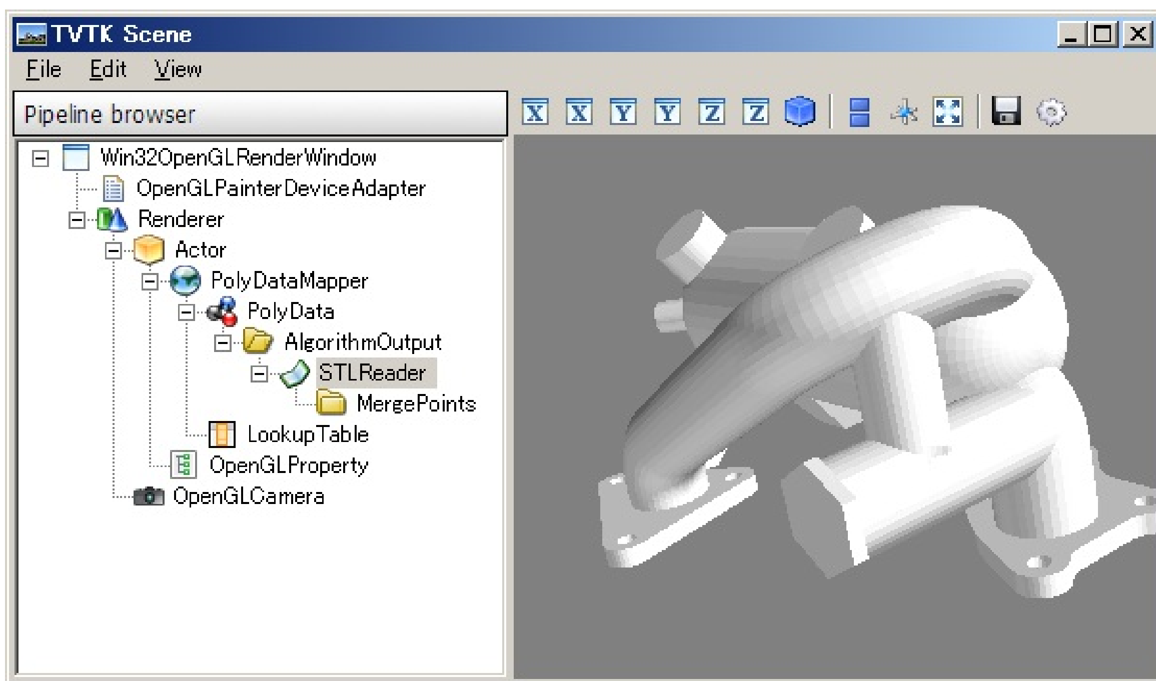
大多数情况下我们不会只用VTK显示圆锥这样的简单物体，VTK的建模功能并不强大，因此它支持许多种格式的文件，能将其它软件产生的数据通过各种Reader类读入VTK，放到流水线上处理。下面的例子从文件42400-IDGH.stl中载入模型数据，并且润色显示。

```
# -*- coding: utf-8 -*-
from enthought.tvtk.api import tvtk
from enthought.tvtk.tools import ivtk
from enthought.pyface.api import GUI

part =tvtk.STLReader(file_name = "42400-IDGH.stl")
part_mapper = tvtk.PolyDataMapper( input = part.output )
part_actor = tvtk.Actor( mapper = part_mapper )

gui = GUI()
window = ivtk.IVTKWithBrowser(size=(800,600))
window.open()
window.scene.add_actor( part_actor )
gui.start_event_loop()
```

此程序的运行画面如下：



显示文件中的3D模型

对比显示圆锥的程序，除了PolyDataMapper的输入从ConeSource改为STLReader之外，其他的部分没有任何区别。STLReader对象知道如何读取STL文件中的数据，并且转换为PolyData，以供PolyDataMapper使用。另外请注意我们这次用ivtk.IVTKWithBrowser产生一个不带Python Shell的ivtk窗口。

STL是什么文件

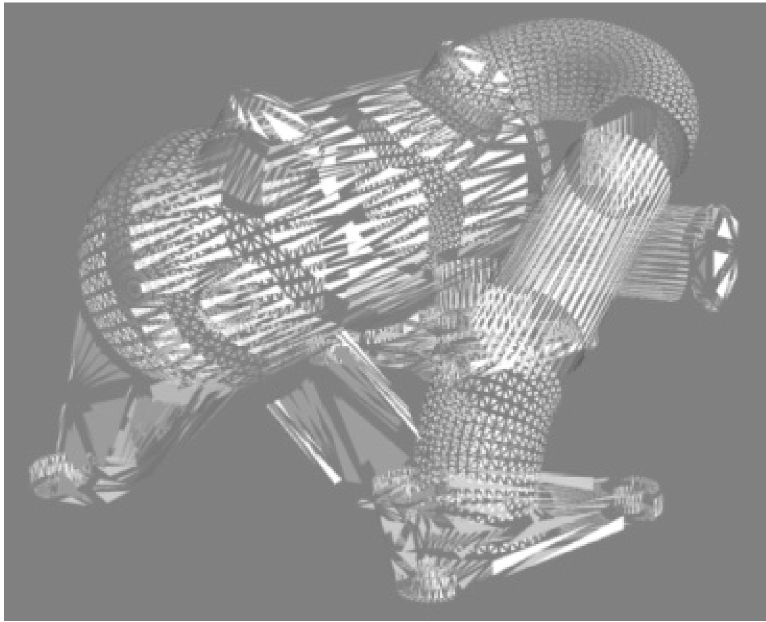
STL的全称为stereo-lithography，由3D Systems公司开发，它使用三角形面片来表示三维实体模型，现已成为CAD/CAM系统接口文件格式的工业标准之一，绝大多数造型系统能支持并生成此种格式文件。例子中的42400-IDGH.stl文件来自于VTK的示例数据。笔者对模具设计没有研究，只是照葫芦画瓢，把VTK的例子转换为TVTK而已。

过滤数据

前面的例子中包括一个数据源和mapper对象，但是流水线中没有过滤器对数据进行过滤，下面我们看看如何对数据进行过滤以减少多边形面的数量。对上节的程序进行修改，在STLReader和PolyDataMapper之间插入一个ShrinkPolyData对象：

```
part =tvtk.STLReader(file_name = "42400-IDGH.stl")
shrink = tvtk.ShrinkPolyData(input = part.output, shrink_factor = (
part_mapper = tvtk.PolyDataMapper( input = shrink.output )
```

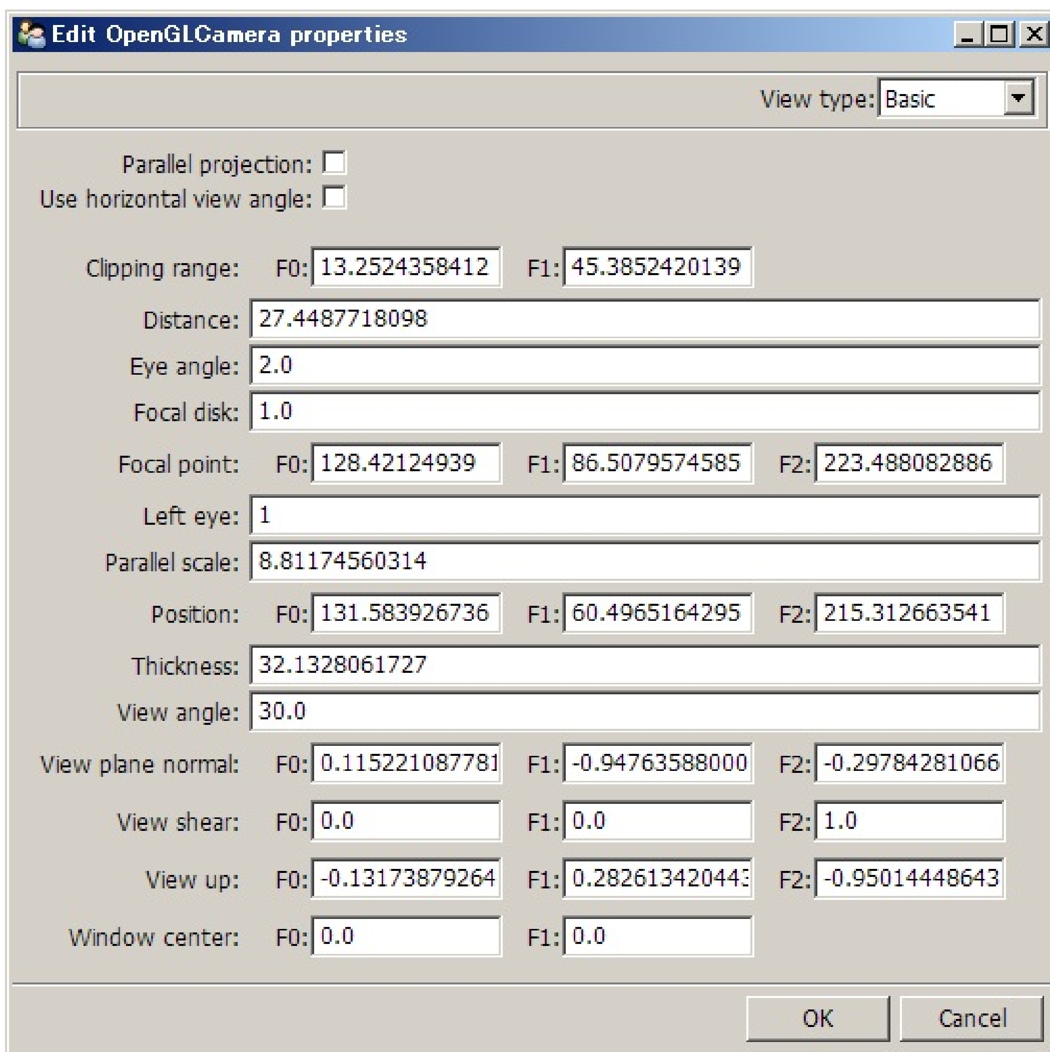
ShrinkPolyData过滤器的输入和输出都是PolyData，它可以减少输入PolyData对象中单元(点线面)的数目，但是会造成不单元之间不连续。



使用ShrinkPolyData过滤器过滤后的模型

控制照相机

如果你使用ivtk显示3D数据的话，在左边的流水线浏览器中可以找到OpenGLCamera，双击它弹出如下窗口：



编辑照相机属性的对话框

这个窗口显示的是3D场景的照相机的所有配置。如果你需要用程序控制照相机的话，可以用：

```
>>> camera = window.scene.renderer.active_camera
```

获得场景中的当前照相机对象，然后就可以获得或者修改照相机的各项配置：

```
>>> camera.clipping_rage
array([ 20.46912341,  51.21854284])
>>> camera.view_up = 0,1,0
```

下面介绍一些照相机的一些常用属性：

- **clipping_plane**：它有两个元素，分别表示照相机到近、远两个裁剪平面的距离。在这两个平面之外的对象将不会被显示出来。
- **position**：照相机在三维空间中的坐标
- **focal_point**：照相机所聚焦的焦点坐标
- **view_up**：照相机的上方向矢量

- **parallel_projection** : 如果为True的话表示采用平行透视，即在3D场景中平行的线投影到2D平面上将仍然是平行的

这些属性虽然可以完全控制照相机的位置和方向，但是实际操作起来并不方便。当将照相机的焦点已经固定好在某个位置上的话，可以通过调用： ..TODO *azimuth* : 以焦点为圆心，沿着纬度线旋转指定角度，即水平旋转，改变其经度

- **elevation** : 沿着经度线方向旋转指定角度，即垂直旋转，改变其纬度

在以焦点为原点的球体坐标系中对照相机进行操作。这两个函数保持view_up属性不变。

控制照明

照明比照相机容易配置得多，假设你运行了ivtk的圆锥的例子的话，直接在窗口下方的命令行中输入：

```
>>> camera = window.scene.renderer.active_camera
>>> light = tvtk.Light(color=(1,0,0))
>>> light.position=camera.position
>>> light.focal_point=camera.focal_point
>>> window.scene.renderer.add_light(light)
```

..TODO 即可在照相机所在处添加一个红色的光源，它的照射方向为朝向focal_point点。如果你设置light的positional属性为True的话，那么它就变成一个探照灯光源，这时照射方向有效。并且可以通过cone_angle属性设置探照灯的光锥角度。光锥为180度的话，就是无方向光源。

控制3D Props

在3D场景中显示的物体通常被称作prop，有几种prop类型，其中包括：Prop3D和Actor。3D场景中所有prop的都从Prop3D继承。

TVTK的改进

下面是使用Python的标准VTK库显示一个圆锥的例子:

```
import vtk

# Source object .
cone = vtk.vtkConeSource( )
cone.SetHeight( 3.0 )
cone.SetRadius( 1.0 )
cone.SetResolution(10)
# The mapper .
coneMapper = vtk.vtkPolyDataMapper( )
coneMapper.SetInput( cone.GetOutput( ) )
# The actor.
coneActor = vtk.vtkActor( )
coneActor.SetMapper ( coneMapper )
# Set it to render in wireframe
coneActor.GetProperty( ).SetRepresentationToWireframe( )

# Renderer and render window .
ren1 = vtk.vtkRenderer( )
ren1.AddActor( coneActor )
ren1.SetBackground( 0.1 , 0.2 , 0.4 )
renWin = vtk.vtkRenderWindow( )
renWin.AddRenderer( ren1 )
renWin.SetSize(300 , 300)

# On screen interaction .
iren = vtk.vtkRenderWindowInteractor( )
iren.SetRenderWindow( renWin )
iren.Initialize( )
iren.Start( )
```

我们可以出这个例子和C++的程序的区别仅仅是没有声明变量的类型，其它的用法完全是按照C++的VTK API调用的。官方所提供的VTK-Python包和C++语言的接口相似，许多地方没有能够体现出Python作为动态语言的优势，可以说标准的VTK-Python库不够Python风格。为了弥补标准库的这些不足之处，Enthought.com开发了TVTK库进一步对VTK进行包装，它具有如下的一些优点:

- 支持Trait属性
- 支持元素的Pickle操作
- API更接近Python风格
- 能自动处理numpy的数组或者Python的列表
- 高级的脚本式的mlab API，和流水线浏览器ivtk
- tvtk的场景和流水线浏览器支持Envisage插件

TVTK的基本用法

下面是用TVTK实现上面的显示圆锥的例子:

```
from enthought.tvtk.api import tvtk

cone = tvtk.ConeSource( height=3.0, radius=1.0, resolution=10 )
cone_mapper = tvtk.PolyDataMapper( input = cone.output )
cone_actor = tvtk.Actor( mapper=cone_mapper )
cone_actor.property.representation = "w"

ren1 = tvtk.Renderer()
ren1.add_actor( cone_actor )
ren1.background = 0.1, 0.2, 0.4
ren_win = tvtk.RenderWindow()
ren_win.add_renderer( ren1 )
ren_win.size = 300, 300

iren = tvtk.RenderWindowInteractor( render_window = ren_win )
iren.initialize()
iren.start()
```

可以看到这个程序比标准VTK版本要简短得多，从中我们可以看到TVTK的一些重要的更改：

- tvtk用 from enthought.tvtk.api import tvtk 语句载入
- tvtk的类名为VTK的类名除去前缀"vtk"。有些类名在"vtk"之后是数字：vtk3DSImporter，由于Python的标示符首字符不能为数字，因此tvtk对此进行特殊处理:如果首字符为数字，则用其英文单词代替: ThreeDSImporter。
- tvtk对象的方法名按照Enthought的一贯用法，采用下划线连接单词：例如如果VTK中的AddItem，将对应tvtk中的add_item。
- 许多VTK的方法在tvtk中用trait属性替代，例如前面的例子中我们用m.input = cs.output表示VTK中的m.SetInput(cs.GetOutput())，p.representation = 'w'表示VTK中的p.SetRepresentationToWireframe()。由于在VTK中许多需要用Set*, Get*的属性在TVTK中都是以Trait属性表现的，Trait属性的初始化可以在产生对象的同时进行配置。
- trait属性可以在创建类的对象的同时通过关键字参数进行设置

在内部实现中，所有的tvtk对象都内部包装有一个VTK对象，对tvtk对象的方法的调用将转给相应的VTK对象的方法执行，如果返回值是VTK对象的话，将被包装成tvtk对象返回。如果方法的参数是tvtk对象的话，其中的VTK对象将传递给VTK的方法。

通过调用tvtk.to_tvtk(p)，可以得到p中所包装的VTK对象

Trait属性

所有的tvtk类都继承于traits.HasStrictTraits，HasStrictTraits规定了它的子类的对象在创建之后不能对不存在的属性进行赋值。

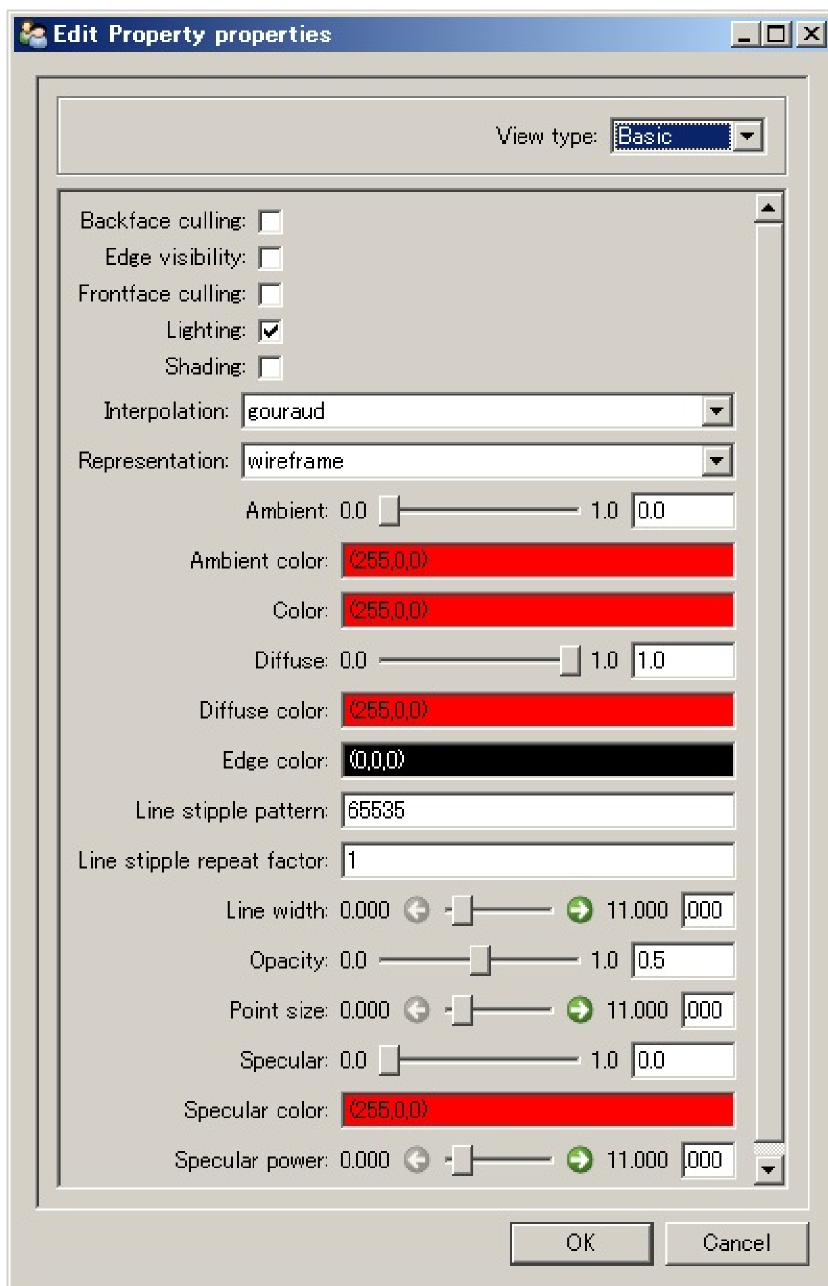
VTK中所有和基本状态有关的的方法在tvtk中都以trait属性表示。trait属性为我们带来如下的便利:

- 通过调用set方法可以一次设置多个trait属性：

```
>>> p = tvtk.Property()
>>> p.set(opacity=0.5, color=(1,0,0), representation="w")
```

- 通过调用edit_traits或者configure_traits方法直接出界面编辑属性。对trait属性的更改将自动作用到内部的VTK对象之上，反过来，内部的VTK对象的状态改变也将自动更新trait属性。下面是一个例子：

```
>>> p.edit_traits()
```



每个TVTK对象都可以有自己的编辑属性的对话框界面

- 我们可以通过tvtk.to_tvtk(p)函数得到任何tvtk对象所包装的TVK对象：

```
>>> print p.representation
wireframe
>>> p_vtk = tvtk.to_vtk(p)
>>> p_vtk.SetRepresentationToSurface()
>>> print p.representation
surface
```

序列化(Pickling)

tvtk对象支持简单的序列化处理。单个tvtk对象的状态可以被序列化：

```
>>> import cPickle
>>> p = tvtk.Property()
>>> p.representation="w"
>>> s = cPickle.dumps(p)
>>> del p
>>> q = cPickle.loads(s)
>>> q.representation
'wireframe'
```

但是序列化仅仅能保存对象的状态，对象之间的引用无法被保存。因此VTK的整个流水线无法用序列化保存。

通常pickle.load将创建新的对象，如果我们希望更新某个已经存在的对象的状态的话，可以如下调用:

```
>>> p = tvtk.Property()
>>> p.interpolation = "flat"
>>> d = p.__getstate__()
>>> del p
>>> q = tvtk.Property()
>>> q.interpolation
'gouraud'
>>> q.__setstate__(d)
>>> q.interpolation
'flat'
```

集合迭代

从tvtk.Collection继承的对象可以像标准的Python序列对象一样使用:

```
>>> ac = tvtk.ActorCollection()
>>> len(ac)
0
>>> ac.append(tvtk.Actor())
>>> ac.append(tvtk.Actor())
>>> len(ac)
2
>>> for a in ac:
...     print a
...
vtkOpenGLActor (06A99EB8)
......
vtkOpenGLActor (069C4270)
......
>>> del ac[0]
>>> len(ac)
1
```

我们看到ActorCollection可以像Python的列表对象一样支持len, append和for循环。对比一下VTK的相应的版本，就能体会出tvtk的好处了:

```
>>> ac = vtk.vtkActorCollection()
>>> ac.GetNumberOfItems()
0
>>> ac.AddItem(vtk.vtkActor())
>>> ac.AddItem(vtk.vtkActor())
>>> ac.GetNumberOfItems()
2
>>> ac.InitTraversal()
>>> for i in range(ac.GetNumberOfItems()):
...     print ac.GetNextItem()
...
vtkOpenGLActor (05E0A750)
......
vtkOpenGLActor (05E0A8C0)
......
>>> ac.RemoveItem(0)
>>> ac.GetNumberOfItems()
1
```

数组操作

所有继承于DataArray类的对象和Python的序列一样，支持迭代接口，以及__getitem__, __setitem__, __repr__, append, extend等等。此外，它还可以直接用numpy的数组或者python的列表直接进行赋值(使用from_array方法)，或者将DataArray中保存的数据转换为numpy的数组。Points和IdList等类也同样支持这些特性：

```
>>> pts = tvtk.Points()
>>> p_array = np.eye(3)
>>> p_array
array([[ 1.,  0.,  0.],
 [ 0.,  1.,  0.],
 [ 0.,  0.,  1.]])
>>> pts.from_array(p_array)
>>> pts.print_traits()
_in_set:                 0
_vtk_obj:                <vtkCommonPython.vtkPoints vtkobject at 0(
actual_memory_size:      1L
bounds:                  (0.0, 1.0, 0.0, 1.0, 0.0, 1.0)
class_name:              'vtkPoints'
data:                    [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, (
data_type:               'double'
...
number_of_points:        3
reference_count:         1
>>> pts.to_array()
array([[ 1.,  0.,  0.],
 [ 0.,  1.,  0.],
 [ 0.,  0.,  1.]])
```

此外tvtk的方法或者属性如果接受DataArray, Points, IdList或者CellArray的对象的话，那么它也同时支持数组和列表:

```
>>> points = np.array([[0,0,0],[1,0,0],[0,1,0],[0,0,1]], 'f')
>>> triangles = np.array([[0,1,3],[0,3,2],[1,2,3],[0,2,1]])
>>> values = np.array([1.1, 1.2, 2.1, 2.2])
>>> mesh = tvtk.PolyData(points=points, polys=triangles)
>>> mesh.point_data.scalars = values
>>> mesh.points
[(0.0, 0.0, 0.0), (1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0)
>>> mesh.polys
<tvtk_classes.cell_array.CellArray object at 0x142D4F60>
>>> mesh.polys.to_array()
array([3, 0, 1, 3, 3, 0, 3, 2, 3, 1, 2, 3, 3, 0, 2, 1])
>>> mesh.point_data.scalars
[1.1000000000000001, 1.2, 2.1000000000000001, 2.2000000000000002]
```

注意CellArray类(mesh的polys属性)的处理有所不同，我们给它传入的是一个二维数组，在内存中保存的却是一维的: array([3, 0, 1, 3, 3, 0, 3, 2, 3, 1, 2, 3, 3, 0, 2, 1])。它的格式是[Cell的数据个数, Cell数据..., Cell的数据个数, Cell数据...]，如下图所示。

CellArray用来描述多边形(Cell)和顶点之间的关系，由于每个Cell可以由不同数量的定点组成，因此内部采用上面所述的形式保存。

TVTK是什么

我们通过from enthought.tvtk.api import tvtk载入tvtk，然后像使用一个模块一样使用它。事实上，我们载入的tvtk并不是一个模块，而是某个类的一个实例。之所以会如此设计，是因为VTK库有近千个类，而TVTK对所有这些类都进行了包装，如果一次性载入这么多类，会极大地影响库的载入速度。

我们载入的tvtk虽然是某个类的实例，但是用起来就和模块一样：

- 通过tvtk对象可以使用所有的TVTK类
- 它不需要载入近千个TVTK类就能支持类名的自动完成
- 只有在真正使用某个TVTK类时，它才会被载入

所有tvtk相关的代码全部都保存在tvtk_classes.zip文件中。而tvtk对象的类在此压缩文件里的tvtk_helper.py中定义。对于TVTK中的每个类，tvtk对象都有一个同名的属性和类相对应。

Mayavi-更方便的可视化

虽然VTK 3D可视化软件包功能强大，Python的TVTK包装方便简洁，但是要用这些工具快速编写实用的三维可视化程序仍然需要花费不少的精力。因此基于VTK开发了许多可视化软件，例如：ParaView、 VTKDesigner2、Mayavi2等等。

Mayavi2完全用Python编写，因此它不但是一个方便实用的可视化软件，而且可以方便地用Python编写扩展，嵌入到用户编写的Python程序中，或者直接使用其面向脚本的API：mlab快速绘制三维图。

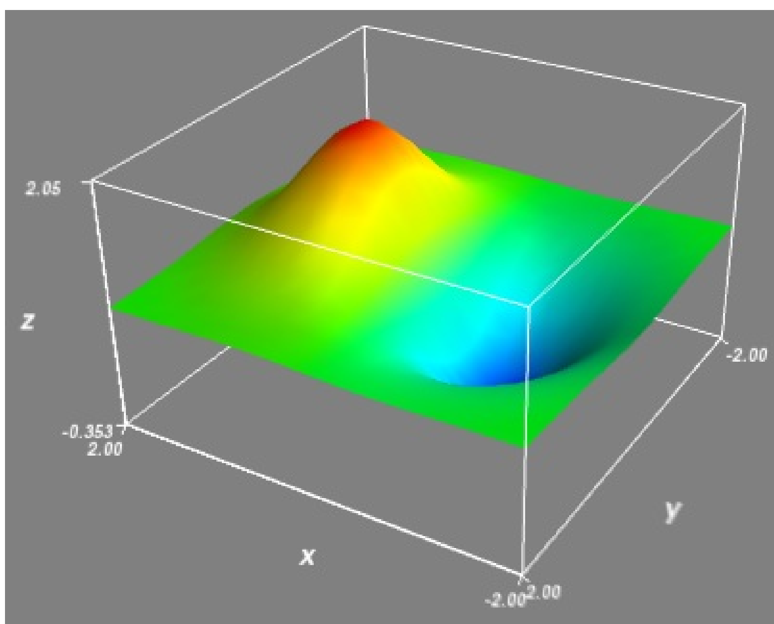
用mlab快速绘图

和Chaco的shell或者matplotlib的pylab一样，mayavi的mlab模块提供了方便快捷的绘制三维图函数。只要把数据准备好，通常只需要调用一次mlab的函数就可以看到数据的三维显示效果。非常适合在IPython中交互式地使用。下面让我们来看一个例子：

```
import numpy as np
from enthought.mayavi import mlab

x, y = np.ogrid[-2:2:20j, -2:2:20j]
z = x * np.exp( - x**2 - y**2)

pl = mlab.surf(x, y, z, warp_scale="auto")
mlab.axes(xlabel='x', ylabel='y', zlabel='z')
mlab.outline(pl)
```



使用Mayavi将二维数组绘制成3D曲面

我们先用下面的语句载入mlab库：

```
from enthought.mayavi import mlab
```

然后通过调用mlab.surf绘制一个三维空间中的曲面。曲面上的每个点的坐标由surf函数的三个二维数组参数x,y,z给出。由于数组x,y是由ogrid对象算出，它们分别是shape为n*1*和*1*n的数组，而z是一个n*n的数组。

通过调用mlab.axes和mlab.outline函数，分别在三维空间中添加坐标轴，和曲面区域的外框。

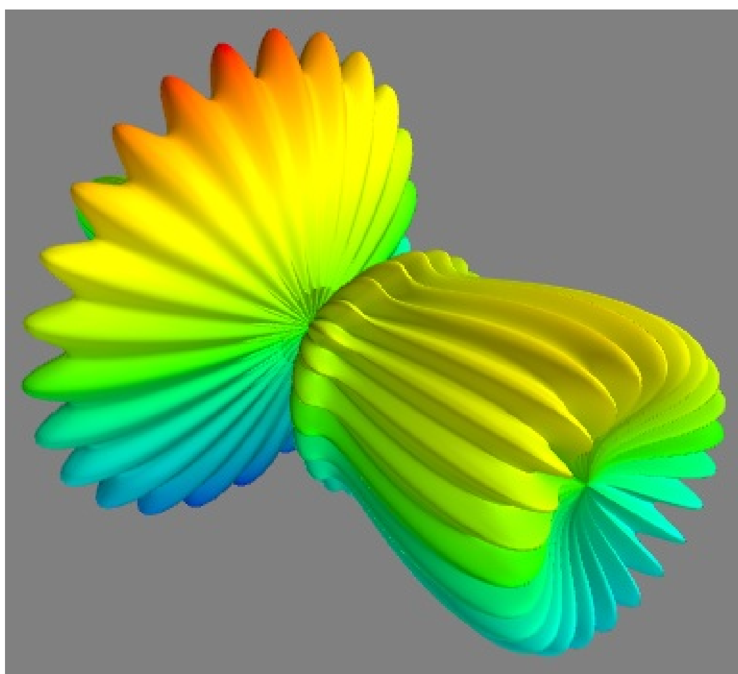
surf绘制的曲面在X-Y平面上的投影是一个等距离的网格，如果需要绘制更复杂的三维曲面的话，可以使用mesh函数。下面是mesh函数的一个例子：

```
# -*- coding: utf-8 -*-
from numpy import *
from enthought.mayavi import mlab

# Create the data.
dphi, dtheta = pi/20.0, pi/20.0
[phi,theta] = mgrid[0:pi+dphi*1.5:dphi,0:2*pi+dtheta*1.5:dtheta]
m0 = 4; m1 = 3; m2 = 2; m3 = 3; m4 = 6; m5 = 2; m6 = 6; m7 = 4;
r = sin(m0*phi)**m1 + cos(m2*phi)**m3 + sin(m4*theta)**m5 + cos(m6
x = r*sin(phi)*cos(theta)
y = r*cos(phi)
z = r*sin(phi)*sin(theta)

# View it.
s = mlab.mesh(x, y, z, representation="wireframe", line_width=1.0 )

mlab.show()
```



使用mesh函数绘制的3D旋转体

mesh和surf类似，其三个数组参数x, y, z也是二维数组，他们相同下标的三个元素组成曲面上某点的三维坐标。点之间的连接关系(边和面)由其在x,y,z数组中间的位置关系决定。

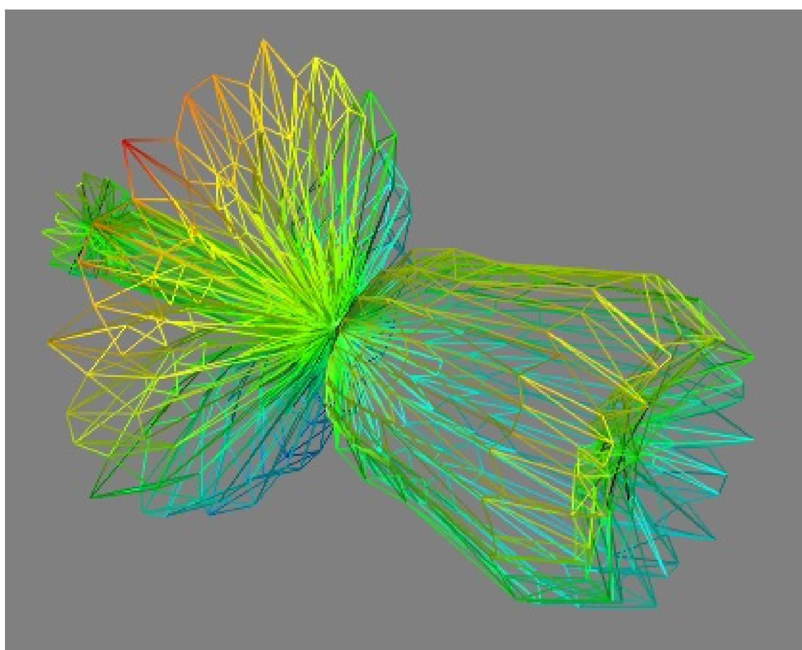
由于这个程序所计算的曲面是一个旋转体，曲面上的各个点的坐标是在球面坐标系中计算的，然后按照坐标转换公式将球面坐标转换为X-Y-Z坐标。

通过传递一个关键字参数representation给mesh函数，可以指定绘制的表现形式：

- **surface**：缺省值，绘制曲面
- **wireframe**：绘制边线，将dphi, dtheta的改为较大值，例如pi/20之后，调用 ：

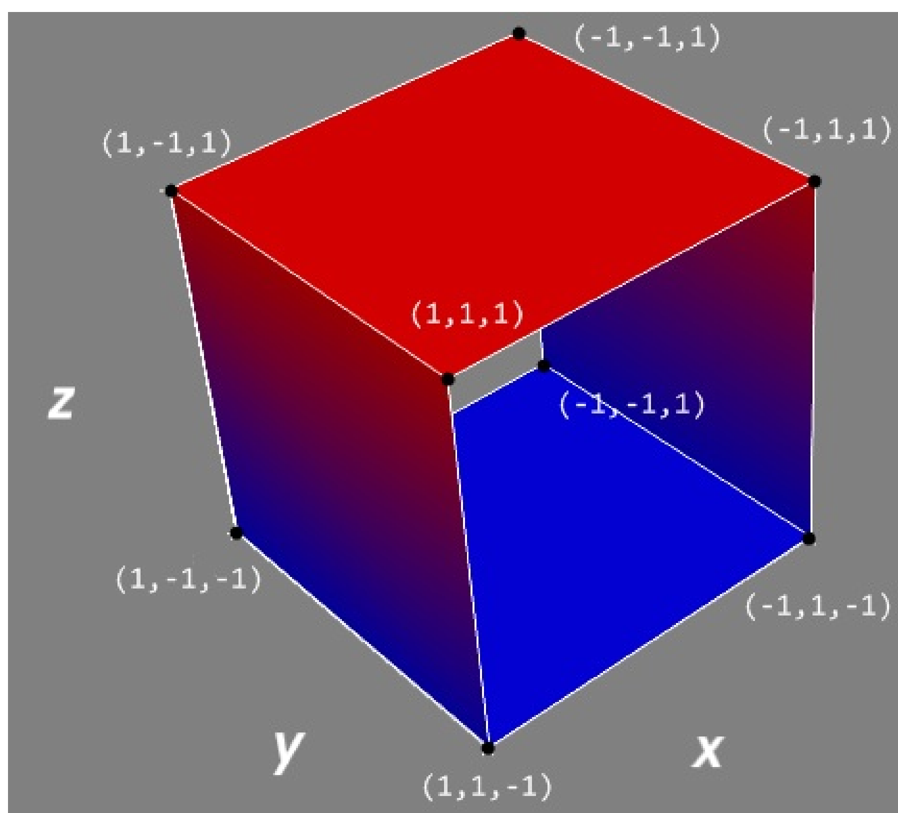
```
s = mlab.mesh(x, y, z, representation="wireframe", line_width=1
```

得到如下结果：



使用mesh函数绘制的线框模型

为了方便理解mesh函数是如何绘制出曲面的，我们通过手工输入坐标的方式，绘制如下图所示的立方体表面的一部分：



组成立方体的各个面和顶点坐标

x,y,z数组的定义如下：

```
x = [[-1,1,1,-1,-1],
[-1,1,1,-1,-1]]

y = [[-1,-1,-1,-1,-1],
[1,1,1,1, 1]]

z = [[1,1,-1,-1,1],
[1,1,-1,-1,1]]
```

x, y, z数组对应坐标的元素组成三维坐标点，因此这三个数组实际描述的坐标点为：

```
[
    [(-1, -1, 1), (1, -1, 1), (1, -1, -1), (-1, -1, -1), (-1, -1, 1
    [(-1, 1, 1), (1, 1, 1), (1, 1, -1), (-1, 1, -1), (-1, 1, 1)]
]
```

点之间的关系有其在数组中的下标决定，因此由：

```
(-1,-1,1),(1,-1,1),(-1,1,1),(1,1,1)
```

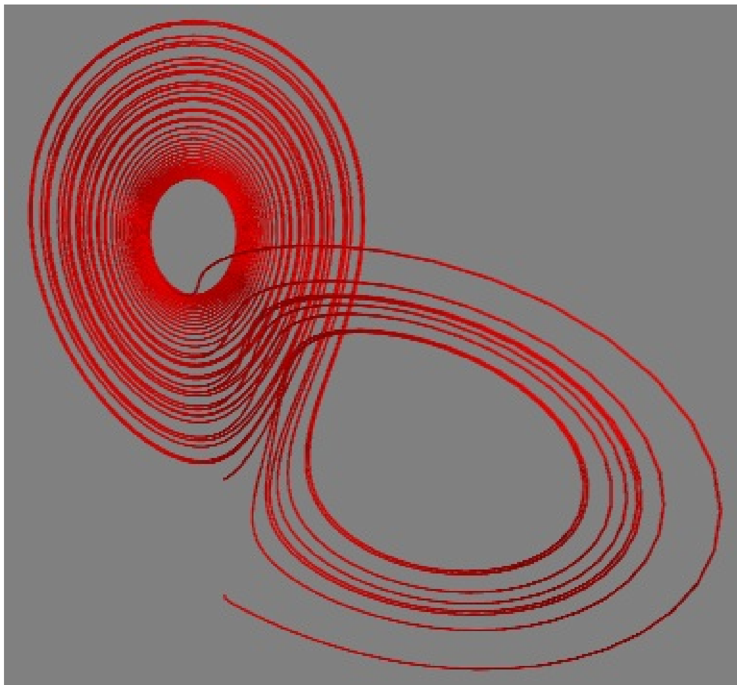
构成一个mesh中的一个面。依次类推，第二个面由：

```
(1,-1,1),(1,-1,-1),(1,1,1),(1,1,-1)
```

构成，一共定义有4个面。

下面详细介绍mlab中提供的绘图函数。

- **points3d**, **plot3d**：给它们传递的3个坐标数组x,y,z都是一维的，因此这两个函数绘制出来的是三维空间中的一系列点(points3d)，或者是一条曲线(plot3d)。下图是采用plot3d绘制的洛仑兹吸引子的轨迹：



plot3d函数绘制的洛仑兹吸引子，曲线使用很细的圆管绘制

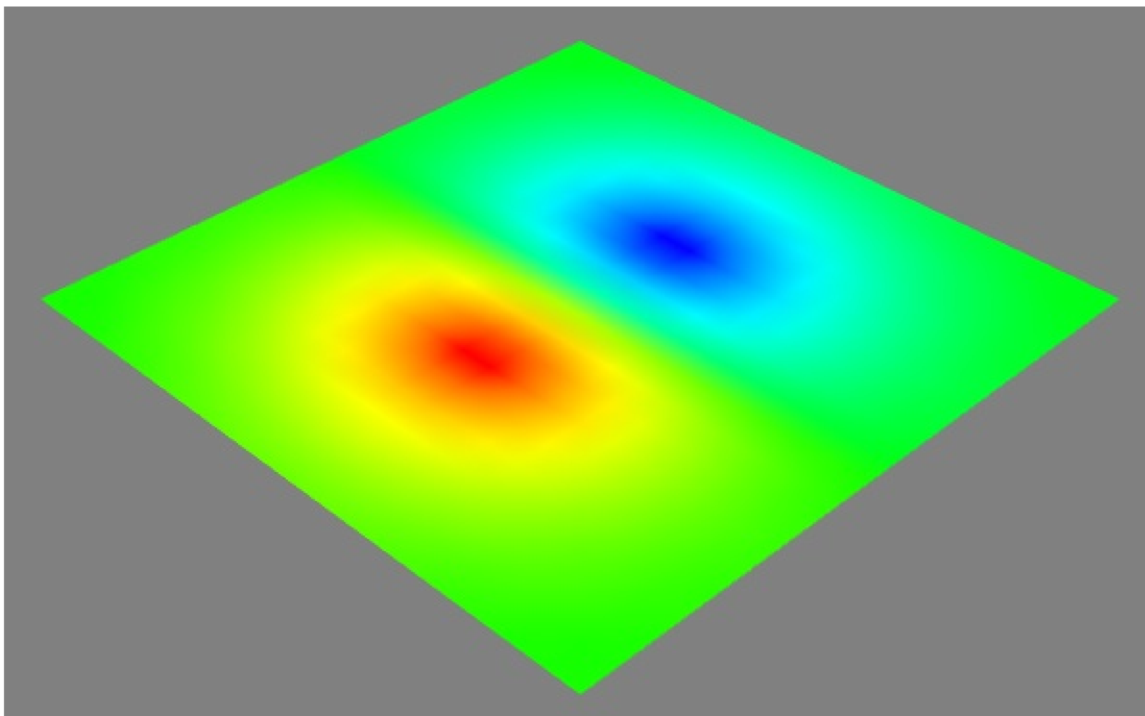
绘图语句的程序如下：

```
mlab.plot3d(track1[:,0], track1[:,1], track1[:,2],color=(1,0,0), tu
```

其中track1为轨迹坐标数组，将其拆分为X,Y,Z轴的三个分量之后，传递给plot3d函数进行绘图。tube_radius指定曲线的粗细，曲线实际上是采用极细的圆管绘制的。

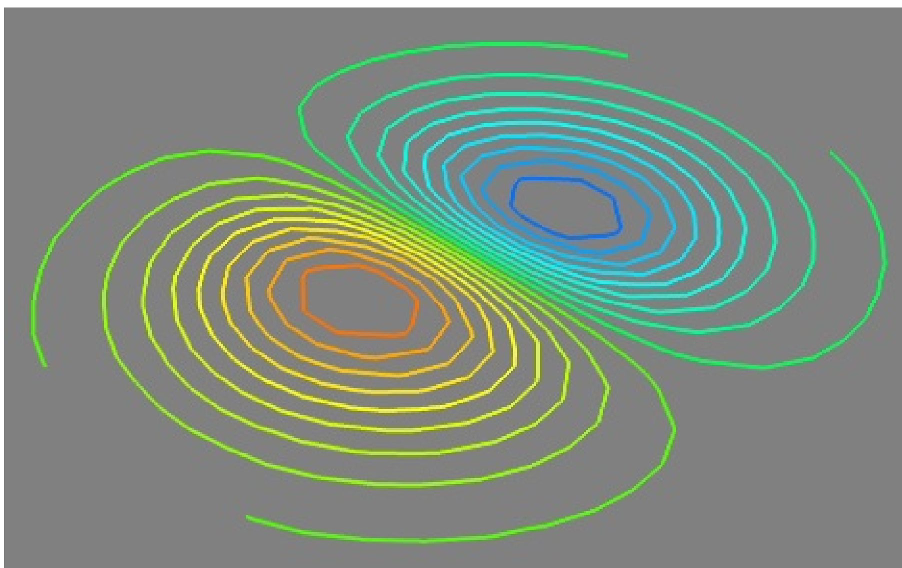
洛仑兹吸引子的轨迹算法请参照：*SciPy-数值计算库*

- **imshow**, **surf**, **contour_surf**：这三个函数都可以接收一个二维数组s，以其第一轴的下标为X轴坐标，第二轴的下标为Y轴坐标。imshow函数将此二维数组当作一个图片显示，每点的颜色为数组s的每个元素的值。surf函数则将此二维数组绘制成三维空间中的曲面，数组中每个元素的值为点的Z轴坐标。contour_surf则绘制二维数组的等高线。下面是imshow函数的绘制结果(所使用的数组和前面surf函数的例子相同)：



imshow函数将二维数组绘制成图像

同样的数据采用contour_surf函数绘制等高线的结果如下图所示：



contour_surf函数绘制二维图像的等高线

Mayavi应用程序

将Mayavi嵌入到界面中

Mayavi除了能够单独作为应用程序使用之外，也可以通过traits属性嵌入到TraitsUI制作的用户应用程序的界面中去，下面的程序演示了这一过程：

```
# -*- coding: utf-8 -*-
from enthought.traits.api import *
from enthought.traits.ui.api import *
from enthought.tvtk.pyface.scene_editor import SceneEditor
from enthought.mayavi.tools.mlab_scene_model import MlabSceneModel
from enthought.mayavi.core.ui.mayavi_scene import MayaviScene

class DemoApp(HasTraits):
    plotbutton = Button(u"绘图")
    scene = Instance(MlabSceneModel, ()) # mayavi场景

    view = View(
        VGroup(
            Item(name='scene',
                editor=SceneEditor(scene_class=MayaviScene), # 设置
                resizable=True,
                height=250,
                width=400
            ),
            'plotbutton',
            show_labels=False
        ),
        title=u"在TraitsUI中嵌入Mayavi"
    )

    def _plotbutton_fired(self):
        self.plot()

    def plot(self):
        g = self.scene.mlab.test_mesh()

app = DemoApp()
app.configure_traits()
```

程序一开始除了从traits和traits.ui库导入之外，还分别从不同的地方导入了SceneEditor、MlabSceneModel和MayaviScene等三个类。

MlabSceneModel类是包装整个mlab的场景的模型，它是属于模型(Model)方面的东西，因此程序中通过:

```
scene = Instance(MlabSceneModel, ())
```

创建一个traits属性scene，使它是MlabSceneModel类的对象。接下来要在视图(View)中创建一个编辑器，让它正确显示scene所代表的模型：

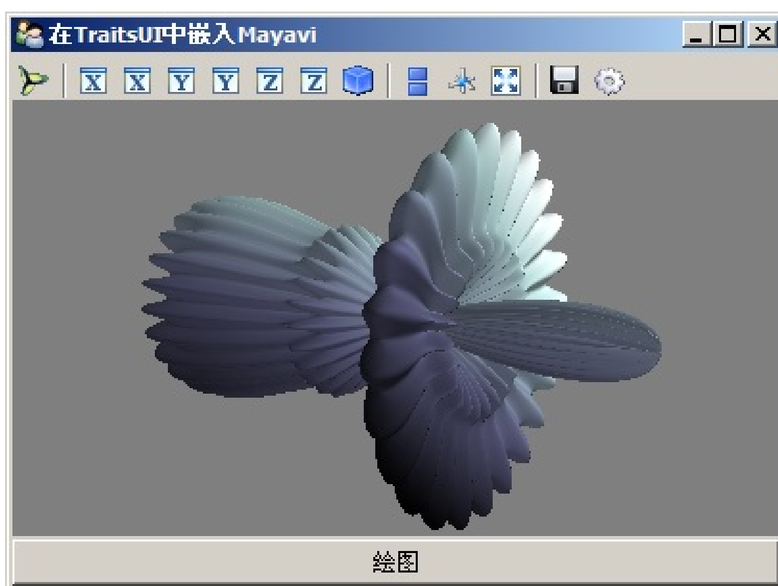
```
Item(name='scene',
    editor=SceneEditor(scene_class=MayaviScene), # 设置mayavi的编辑器
    resizable=True,
    height=250,
    width=400
)
```

SceneEditor是用来创建场景编辑器的工厂类，通过关键字scene_class指定真正创建场景对象类MayaviScene。

程序中我们还创建了一个plotbutton按钮，当此按钮被按下时，调用_plotbutton_fired函数，从而调用最后的绘制场景的函数plot，plot函数只有一句话：

```
g = self.scene.mlab.test_mesh()
```

scene.mlab和前面所介绍的mlab库一样使用，我们调用其test_mesh测试函数，快速在scene中创建一个如下图所示的很酷的曲面体。



将Mayavi嵌入到TraitsUI制作的界面中

下面让我们来看一个有些实用价值的程序，用户输入一个使用x,y,z等变量的函数f(x,y,z)，例如xx+yy+z*z，程序将使用此函数计算一个指定坐标范围之内的三维标量场。并且添加等值面和切面两个工具观察此标量场。等值面可以是自动计算，或者通过滚动条手工配置；而切面的位置和方向则可以直接在场景中用鼠标进行操作。完整的程序请参考三维标量场观察器，下面对程序中的重点部分进行说明。

用户点击描绘按钮之后，调用plot函数绘图，plot函数中首先计算三维标量场，注意我们使用mgrid快速产生三维网格，x0, x1, y0, y1, z0, z1, points, function等都是traits属性，可以通过界面直接修改其值：

```
# 产生三维网格
x, y, z = mgrid[
    self.x0:self.x1:1j*self.points,
    self.y0:self.y1:1j*self.points,
    self.z0:self.z1:1j*self.points]
scalars = eval(self.function) # 根据函数计算标量场的值
```

然后清空当前的场景：

```
self.scene.mlab.clf() # 清空当前场景
```

接下来调用scene.mlab中的axes, contour3d, pipeline.scalar_cut_plane等函数在场景中添加等值面、坐标轴和切面：

```
# 绘制等值面
g = self.scene.mlab.contour3d(x, y, z, scalars, contours=8, transpa
g.contour.auto_contours = self.autocontour
self.scene.mlab.axes() # 添加坐标轴

# 添加一个X-Y的切面
s = self.scene.mlab.pipeline.scalar_cut_plane(g)
cutpoint = (self.x0+self.x1)/2, (self.y0+self.y1)/2, (self.z0+self.
s.implicit_plane.normal = (0,0,1) # x cut
s.implicit_plane.origin = cutpoint
```

最后更新几个属性，其中v0和v1是标量场的最小和最大值，用来设置等值面滚动条的取值范围：

```
self.g = g
self.scalars = scalars
# 计算标量场的值的范围
self.v0 = np.min(scalars)
self.v1 = np.max(scalars)
```

当界面中的“自动等值”选项值(autocontour属性)改变时，通过调用如下程序改变场景中的等值面的自动选项：

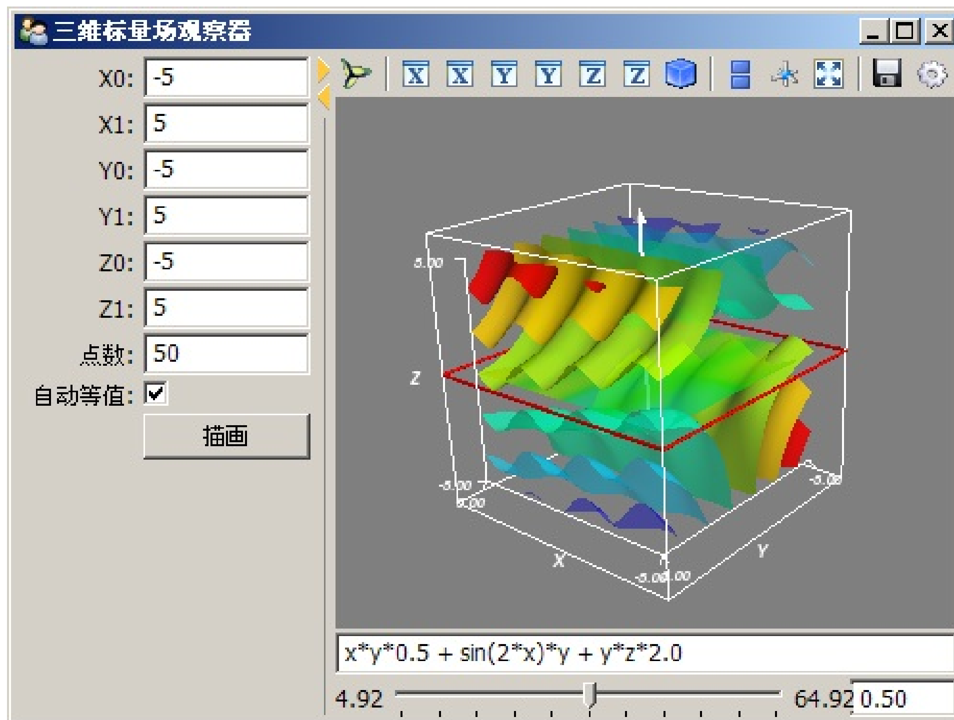
```
self.g.contour.auto_contours = self.autocontour
```

当界面中的滚动条值(contour属性)发生变化时，通过下面的程序修改场景中等值面的值:

```
if not self.g.contour.auto_contours:
    self.g.contour.contours = [self.contour]
```

剩下的部分都是使用标准的traits和TraitsUI库编写的，请参考*TraitsUI*-轻松制作用户界面，这里就不再多解释了。

此程序的界面截图如下图所示：



三维标量场观察器： `x*y*0.5+2*y*sin(2*x)+y*z*2.0`

Visual-制作3D演示动画

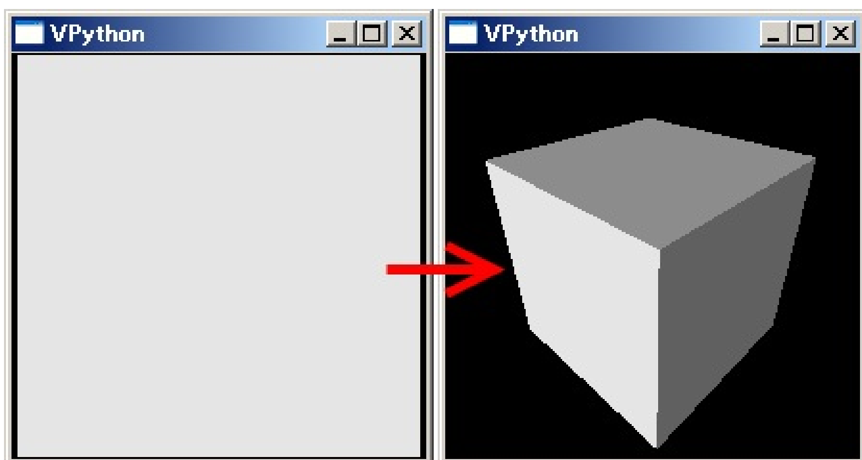
Visual 是Python的一个简单易用的3D图形库，使用它可以快速创建3D场景、动画。和TVTK相比它更加适合于创建交互式的3D场景，而TVTK则更加适合于数据的3D图形化显示。在本节中将通过一个实例简单的介绍如何使用Visual制作3D动画。

场景、物体和照相机

先来看一个最简单的例子：

```
from visual import *
box()
```

这个程序的运行结果如下图的左图所示：



用鼠标旋转之后，可以看出VPython绘制的立方体

我们先从visual库中载入所有对象，然后通过box()创建一个box类的实例，创建这个实例的同时将产生一标题为VPython的场景窗口。由于我们没有给box传递参数，所创建的立方体的所有属性都是缺省配置：

- 立方体的3D空间的坐标为 0, 0, 0，即坐标原点
- 立方体的大小为1, 1, 1
- 立方体的颜色为白色

而场景中的照相机缺省从Z轴的上方往下看（俯视图），缩放比例缺省是正好显示场景中的所有物体。于是我们在场景中看到的是一个刚好充满场景窗口的正方形。

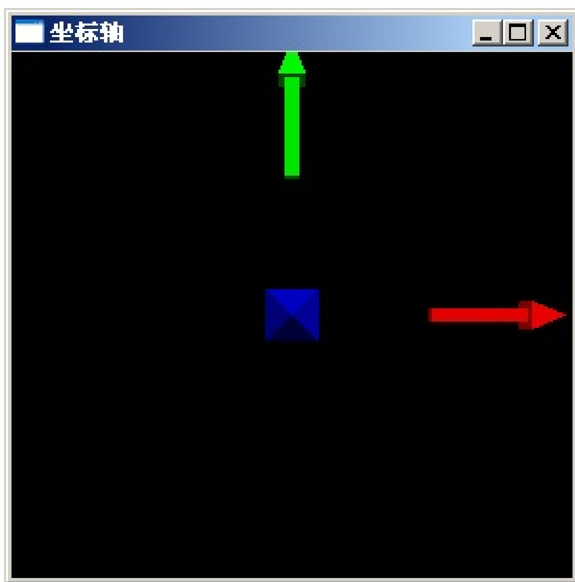
照相机

照相机实际上就是我们观察3D场景的工具，我们通过照相机观察场景中的物体，照相机本身在场景中是不可见的。缩放比例和旋转场景其实都是对照相机进行操作，进行这些操作时，场景中的物体并没有改变，只是我们观察物体的方位改变了。

在场景窗口中，同时按住鼠标左右按键，上下移动鼠标可以进行缩放场景；按住鼠标右键移动鼠标可以旋转场景。右图是进行适当的旋转和缩放之后的效果。我们看到box()确实是创建了一个立方体。

为了搞清楚照相机的位置和坐标轴之间的关系，让我们运行下面这个小程序：

```
# -*- coding: utf-8 -*-
from visual import *
display(title=u"坐标轴".encode("gb2312"), width=300, height=300)
arrow(pos=(1,0,0), axis=(1,0,0), color=(1,0,0))
arrow(pos=(0,1,0), axis=(0,1,0), color=(0,1,0))
arrow(pos=(0,0,1), axis=(0,0,1), color=(0,0,1))
```



VPython照相机的缺省位置，红绿蓝分别表示X,Y,Z轴

这段程序中，我们通过调用display()创建一个场景窗口，并且指定了窗口的标题、宽度和高度。标题必须使用Windows系统缺省的编码，因此为了显示中文，需要将unicode转换为gb2312编码。

调用3次arrow()创建了三个箭头物体，我们通过几个关键字参数配置箭头的属性：

- 箭头的起点坐标用pos关键字参数指定，分别为(1,0,0), (0,1,0), (0,0,1)，坐标用3元组元表示。这三个坐标都在坐标轴上。
- 箭头的方向和长度使用axis关键字参数指定，其值为3D空间的矢量，矢量也是用三元组元表示，程序中所用的三个矢量正好是三个坐标轴的方向，长度为1。
- 通过color参数指定箭头物体的颜色，颜色也是用三元组元表示，取值范围为0到1，分别表示红、绿、蓝三色的成分。

通过观察图中的三个箭头的位置，我们可以知道：

- 窗口的中心为坐标原点
- x轴为从左到右
- y轴为从下到上

- z轴为从屏幕里到屏幕外

因此此时的照相机位于z轴正方向上的某点，方向沿着z轴负方向俯视。

简单动画

下面让我们来看看如何用visual创建一个简单的3D动画，先看一下完整的程序：

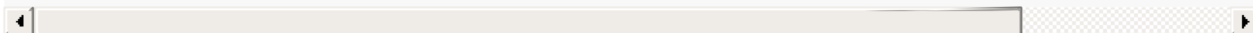
```
# -*- coding: utf-8 -*-
from visual import *

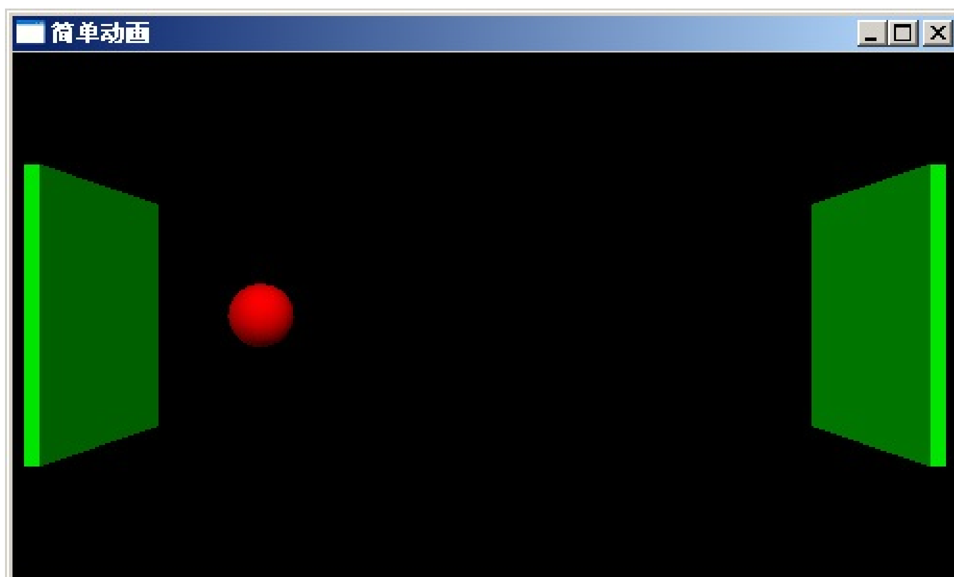
display(title=u"简单动画".encode("gb2312"), width=500, height=300)

ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red)
wall_right = box(pos=(6,0,0), size=(0.1, 4, 4), color=color.green)
wall_left  = box(pos=(-6,0,0), size=(0.1, 4, 4), color=color.green

dt = 0.05
ball.velocity = vector(6, 0, 0)

while True:
    rate(1/dt)
    ball.pos = ball.pos + ball.velocity*dt
    if ball.x > wall_right.x-ball.radius or ball.x < wall_left.x+ba
        ball.velocity.x *= -1
```





球在板子之间反复运动的简单动画

运行这段程序会出现一个有两块绿色板子和一个红球的窗口，红球在两块板子之间反复运动。

第6-8行创建了场景中的三个物体：两块绿色的板子(box)和一个红色的球(sphere)。sphere可以通过radius属性设置其半径，而box可以通过size属性设置其x, y, z轴方向的长度。前面提到过axis属性也可以改变box的大小，这两个属性是互相影响的，在用户手册中我们会详细讨论这个问题。

第10行定义了一个变量dt，我们用它来表示动画中每帧之间的时间间隔。第11行我们给ball添加一个velocity属性，它是一个3D矢量表示球体的速度。请注意velocity不是sphere类固有的属性，是我们为ball物体动态添加的属性。

第13行开始一个死循环，在这个循环中不断地更新ball的pos属性以实现动画效果，为了控制动画的播放速度，在循环中先调用rate函数。由于dt为0.05秒，因此我们动画速度为每秒20帧。rate函数会让程序等待足够长的时间使得动画播放的帧数接近指定的帧数。

第15行修改ball的pos属性，加上在dt时间段中ball的位移量。第16, 17行处理和板子的碰撞，因为pos为球的中心坐标，而碰撞点在球的表面，因此处理碰撞时还需要考虑球的半径。确定碰撞之后，只需要将球的速度反转即可。

由于球的速度为6，而两板之间的间隔为12，因此球从左板移动到右板需要2秒钟时间。

盒子中反弹的球

下面让我们来看一个完整的反弹动画程序。在场景中放置6个半透明的墙面，形成一个正方体，球体的在正方体内部运动反弹，我们可以调整重力加速度(Z方向的加速度)和反弹系数，同时还显示球的速度矢量和运动轨迹。下面是完整的程序：

```
# -*- coding: utf-8 -*-
from visual import *

display(title=u"简单动画".encode("gb2312"), width=500, height=500)

# 创建球体和6个墙面，墙面设置为半透明，以观察球体的运动轨迹
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red)
wall_right = box(pos=(6,0,0), size=(0.1, 12, 12), color=color.green
wall_left  = box(pos=(-6,0,0), size=(0.1, 12, 12), color=color.gree
wall_front = box(pos=(0,-6,0), size=(12, 0.1, 12), color=color.gree
wall_back  = box(pos=(0,6,0), size=(12, 0.1, 12), color=color.green
wall_bottom = box(pos=(0,0,-6), size=(12, 12, 0.1), color=color.gre
wall_top   = box(pos=(0,0,6), size=(12, 12, 0.1), color=color.green

dt = 0.05
g = 9.8 # 重力加速度
f = 0.9 # 反弹能量保持系数，1.0表示完全反弹
ball.velocity = vector(8, 6, 12)
bv = arrow(pos = ball.pos, axis=ball.velocity*0.2, color=color.yell
ball.trail = curve(color=ball.color)
trail_color = 0 # 轨迹的颜色

while True:
```

```
    rate(1/dt)

    # 重力加速度改变z轴方向的速度，不存在反弹时修改速度
    ball.velocity.z -= g * dt

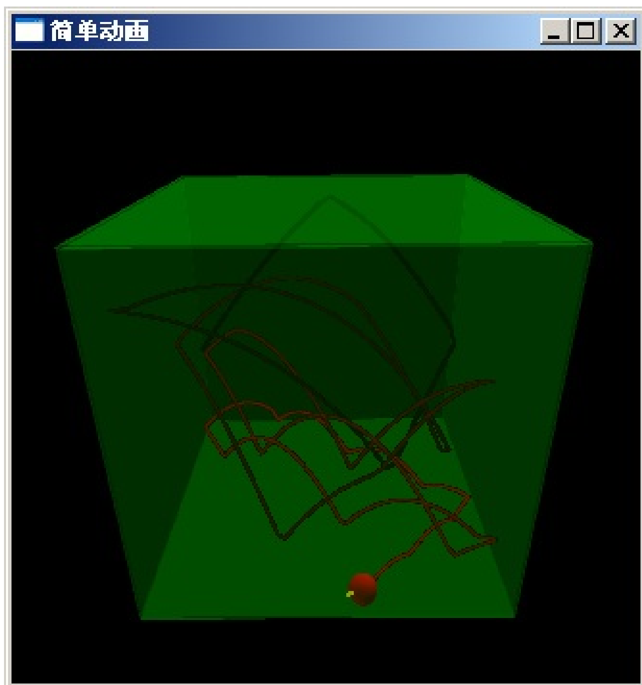
    # 根据速度修改球体的位置
    ball.pos += ball.velocity * dt

    ## 速度为正时判断正方向的墙，速度为负时判断负方向的墙
    ## 处理反弹时需要修正球的位置，使它正好和墙面接触
    # 处理左右墙的反弹
    if ball.velocity.x > 0 and ball.x >= wall_right.x - ball.radius
        ball.x = wall_right.x - ball.radius
        ball.velocity.x *= -f
    if ball.velocity.x < 0 and ball.x <= wall_left.x + ball.radius
        ball.x = wall_left.x + ball.radius
        ball.velocity.x *= -f

    # 处理前后墙的反弹
    if ball.velocity.y > 0 and ball.y >= wall_back.y - ball.radius
        ball.y = wall_back.y - ball.radius
        ball.velocity.y *= -f
    if ball.velocity.y < 0 and ball.y <= wall_front.y + ball.radius
        ball.y = wall_front.y + ball.radius
        ball.velocity.y *= -f

    # 处理上下墙的反弹
    if ball.velocity.z > 0 and ball.z >= wall_top.z - ball.radius:
        ball.z = wall_top.z - ball.radius
        ball.velocity.z *= -f
    elif ball.velocity.z < 0 and ball.z <= wall_bottom.z + ball.rad
        ball.z = wall_bottom.z + ball.radius
        ball.velocity.z *= -f

    # 更新速度箭头的位置和方向
    bv.pos = ball.pos
    bv.axis = ball.velocity*0.2
    # 添加球的轨迹点
    ball.trail.append( pos = ball.pos, color = (trail_color, 0, 0)
    trail_color += 1.0/30.0*dt # 30秒后颜色变为全红
    if trail_color > 1.0: trail_color = 1.0
```



球在封闭的盒子中反弹的动画

第8-13行创建上下左右前后六个墙面，通过设置其opacity属性，设置其不透明度为0.2。opacity=0.0表示完全透明，opacity=1.0表示完全不透明。

第19行用arrow()创建了一个箭头物体，它的起始点位置为球体的中心，方向和球体的速度方向相同：

```
bv = arrow(pos = ball.pos, axis=ball.velocity*0.1, color=color.yel
```

第20行用cureve()创建一个曲线物体，并赋值给球体的trail属性：

```
ball.trail = curve(color=ball.color)
```

第27行使用加速度更新球体的速度，第30行使用速度更新球的体位移。

第35-56行，处理球体和墙壁的碰撞，x, y, z三个方向的碰撞处理方式相同，这里以x方向为例简要说明一下碰撞处理。

当球体的x轴方向的速度为正时，判断球体是否和正方向的墙壁(右墙)相撞，如果相撞的话则将其x轴方向的速度反向，并且乘以碰撞系数模拟能量损失，同时修改球体的x轴坐标，使得其正好和右墙相接触。球体的x轴方向速度为负时，和左墙进行碰撞检测：

```
if ball.velocity.x > 0 and ball.x >= wall_right.x - ball.radius:
    ball.x = wall_right.x - ball.radius
    ball.velocity.x *= -f
if ball.velocity.x < 0 and ball.x <= wall_left.x + ball.radius:
    ball.x = wall_left.x + ball.radius
    ball.velocity.x *= -f
```

第59,60行更新箭头物体的位置和方向以表示球体的速度。第62行将现在的球体的位置添加进球体的轨迹曲线物体。第63,64行更新轨迹的颜色，这样颜色按照随着时间逐渐变红，从黑变红一共需要30秒时间。

OpenCV-图像处理和计算机视觉

OpenCV是Intel公司开发的开源计算机视觉库。它用C语言高速地实现了许多图像处理和计算机视觉方面的通用算法，并且通过SWIG提供了Python的调用接口。本章介绍用Python调用OpenCV库，实现一些简单的图像处理和计算机视觉算法。

OpenCV提供的Python调用接口和C语言的API基本上是一致的，这个接口对于动态语言Python来说有些累赘。不过由于Python程序和C语言程序差别不大，用Python调用OpenCV，能够帮助我们测试API函数和快速实现算法。

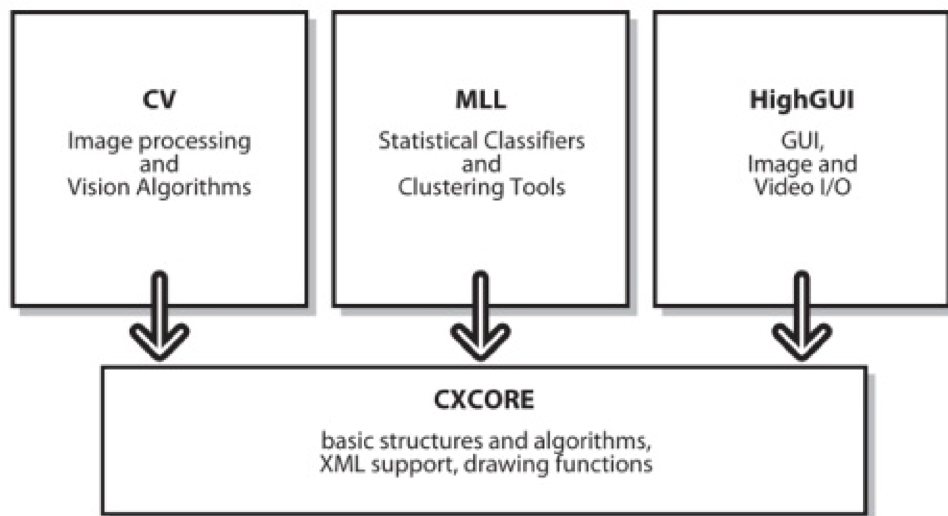
读写图像和视频文件

让我们从显示一幅图像开始进入OpenCV：

```
# -*- coding: utf-8 -*-
from opencv.highgui import *
import sys

img = cvLoadImage( sys.argv[1] )
cvNamedWindow("Example1", CV_WINDOW_AUTOSIZE)
cvShowImage("Example1", img)
cvWaitKey(0)
```

OpenCV的库可以分为5个主要组成部分，下图显示了其中的4个：



OpenCV的5个主要组成部分

- **CV**：包括了基本的图像处理和高级的计算机视觉算法，在Python中，opencv.cv模块与之对应
- **ML**：机器学习库，包括许多统计分类器，opencv.ml模块与之对应
- **HighGUI**：提供各种图像、视频、数据的输入输出和简单的GUI开发，

opencv.highgui模块与之对应
- **CXCore** : 上述三个库都是以CXCore提供的基本数据结构和函数为基础，主模块opencv与之对应
- **CvAux** : 包括一些实验性的算法

显示图像的例子中，只用到数据输入和界面显示两个功能，他们都在highgui库中，因此需要从库中载入这些函数，由于opencv的所有API函数都以cv开头，因此不怕他们和别的库命名冲突：

```
from opencv.highgui import *
```

下面调用cvLoadImage从文件中读入图片信息，其返回的是一个opencv.cv.cvMat对象，cvMat是OpenCV中描述矩阵(或者说多维数组)的数据结构，许多图像处理操作都是针对cvMat对象进行的：

```
img = cvLoadImage( sys.argv[1] )
```

下面调用cvNamedWindow函数创建一个窗口，其名字为"Example1"，大小设置为CV_WINDOW_AUTOSIZE，表示它随着其内容自动改变大小：

```
cvNamedWindow("Example1", CV_WINDOW_AUTOSIZE)
```

然后调用cvShowImage函数，将img表示的图像显示在"Example1"窗口。由于OpenCV库大部分代码都是使用C语言编写的，因此它采用“对象.方法()”的方式，而是使用函数的方式。而且highgui提供的仅是简便的GUI功能，因此这里直接用字符串"Example1"表示要显示图片的窗口，而不是用某个表示窗口的对象。

最后调用cvWaitKey，等待用户按键输入，如果其参数为正值，那么等待指定的毫秒数后继续运行；如果其值为0，表示永久等待：

```
cvWaitKey(0)
```

如果在IPython中运行上面程序之后，IPython等待用户按键输入，按任意键之后，IPython进入可输入命令的状态，并且显示图片的窗口并没有关闭，这样就可以在IPython中直接输入opencv的函数调用，查看其结果。下面的先从opencv.cv载入所有图像处理相关的函数，

```
>>> from opencv.cv import *
```

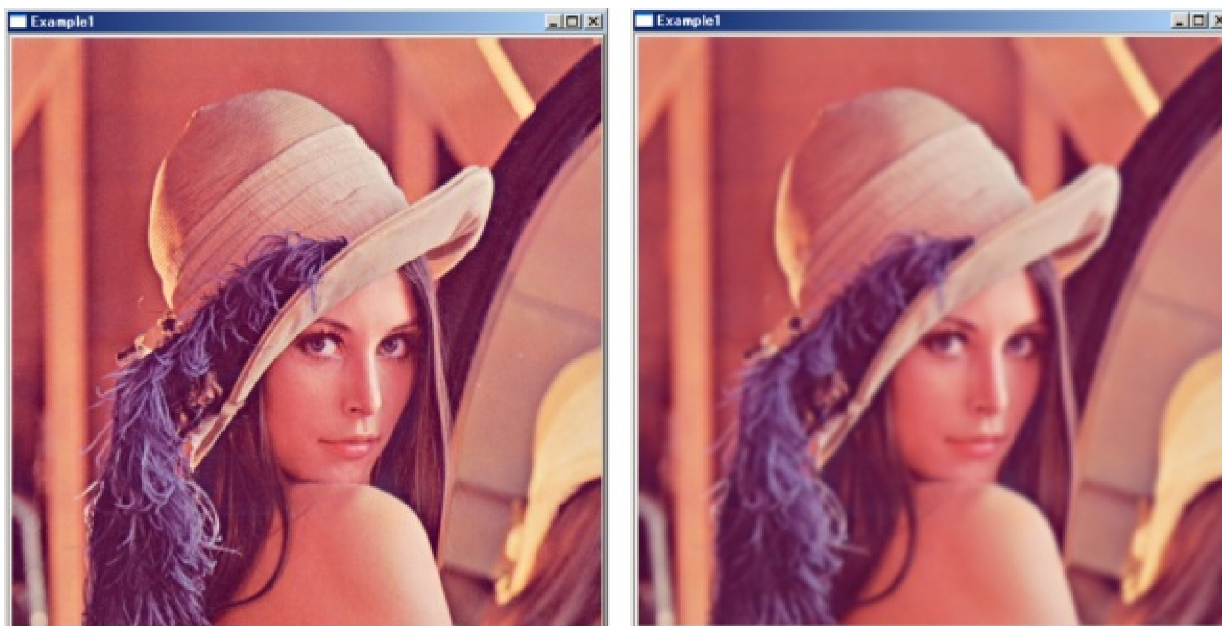
然后调用cvSmooth函数对img进行高斯模糊，cvSmooth函数的第一个参数指定原始图像，第二个参数指定输出图像，这里都用img，因此高斯模糊的结果覆盖原始图像，第三个参数指定采用高斯模糊算法，第四个参数是高斯模糊的参数：以像素点为单位的模糊范围：

```
>>> cvSmooth(img, img, CV_GAUSSIAN, 11)
```

最后调用cvShowImage更新窗口中的图片：

```
>>> cvShowImage("Example1", img)
```

下面是图像处理的结果，左图为原始图像，右图为模糊后的图像：



调用cvSmooth对图像进行高斯模糊处理

Traits使用手册

traits

traits.ui

定义Traits

在Python程序中按照下面的步骤使用Traits库：

1. 从 enthought.traits.api 中载入你所需要的对象
2. 定义你想使用的traits
3. 从HasTraits类继承一个新类，在其中使用你定义的traits声明trait属性

通常第2、3步是放在一起的，也就是说定义traits的同时定义trait属性，在本手册中的大部分例子都是采用这种方式：

```
from enthought.traits.api import HasTraits, Float

class Person(HasTraits):
    weight = Float(50.0)
```

这段程序定义了一个从HasTraits类继承的Person类，在其内部声明了一个名为weight的trait属性，其类型为浮点数，初始值为50.0。trait属性像类的属性一样定义，像实例的属性一样使用。下面我们来看看如何使用trait属性：

```
>>> joe = Person()
>>> joe.weight
50.0
>>> joe.weight = 70.5
>>> joe.weight = 70
>>> joe.weight = "89"
Traceback (most recent call last):
  File "...trait_handlers.py", line 175, in error value )
TraitError: The 'weight' trait of a Person instance must be a float
but a value of '89' <type 'str'> was specified.
```

由于joe是Person类的实例，因此它有一个名为weight的trait属性，并且初始值为50.0。由于weight是使用Float声明的，我们能将浮点数赋值给它，由于整数可以不丢失精度的转换为浮点数，因此整数也可以赋值给它。然而，把浮点数赋值给整数trait属性将会产生错误。由于字符串无法转换为浮点数，因此赋值为字符串产生错误，错误的提示信息告诉我们它需要浮点数。

有时候我们希望trait属性能够自动的进行强制类型转换，这样我们就可以将字符串赋值给类型为float的trait属性，省去了手工转换的麻烦。这种强制类型转换的trait属性都用Casting trait声明，所有的Casting trait都是以 **C** 开头的：

```
from enthought.traits.api import HasTraits, CFloat

class Person(HasTraits):
    cweight = CFloat(50.0)
```

```
>>> bill = Person()
>>> bill.cweight = "90"
>>> bill.cweight
90.0
>>> bill.cweight = "abc"
Traceback (most recent call last)
...
```

这段程序用CFloat声明了一个强制类型转换的trait属性cweight。我们可以将能转换为浮点数的字符串"90"赋值给它，但是 "abc" 这样的字符串赋值仍然会抛出异常。我们可以想象CFloat的内部处理：它先将传入的值用内部函数float()进行强制类型转换，然后把结果赋值给trait属性。

我们也可以先单独定义一个traits，然后用它来声明多个类的多个trait属性，下面是一个例子：

```
from enthought.traits.api import HasTraits, Range

coefficient = Range(-1.0, 1.0, 0.0)

class quadratic(HasTraits):
    c2 = coefficient
    c1 = coefficient
    c0 = coefficient
    x  = Range(-100.0, 100.0, 0.0)
```

在这个例子中，我们需要定义多个trait属性，其类型都为Range(具有取值范围的浮点值)，并且范围都是-1.0到1.0，初始值为0.0。为了体现代码重用，我们先用coefficient = Range(-1.0, 1.0, 0.0)定义了一个traits，然后在quadratic类中使用它定义三个trait属性：c0, c1, c2。

预定义的Traits

Traits库为Python的许多数据类型提供了预定义的trait类型。HasTraits派生的类中用trait类型名直接定义trait属性，这个类的所有实例都将拥有一个初始化为缺省值的属性，例如：

```
class Person(HasTraits):
    age = Float
```

上面的例子为Person类定义了一个age属性，其类型为浮点数，并且被初始化为0.0(Float的缺省值)。如果你希望用别的值初始化trait属性的话，可以把这个值当作参数传递给trait类型：

```
age = Float(10.0)
```

几乎所有的trait类型都是可以带括号调用的，它可以接受缺省值或者其它的参数；还可以通过关键字参数接受元数据。 .. TODO:: 插入元数据链接

简单类型

对于每个Python的简单数据类型都对应两种trait类型：强制类型和自动转换类型。它们的区别在于：

- **强制型Trait**：当这样trait属性被赋值为类型不匹配的数据时，会产生错误
- 自动型**Trait**：类型不匹配时会自动调用此类型对应的Pyton内置的转换函数进行类型转换

强制型 **Trait**	自动型 **Trait**	**Python**对应的数据类型	内置缺省值	自动转换函数
Bool	CBool	Boolean	False	bool()
Complex	CComplex	Complex number	0+0j	complex()
Float	CFloat	Floating point number	0.0	float()
Int	CInt	Plain integer	0	int()
Long	CLong	Long integer	0L	int()
Str	CStr	String	''	str()
Unicode	CUnicode	Unicode	u''	unicode()

下面的例子演示了强制型Trait和自动型Trait之间的区别：

```
>>> from enthought.traits.api import HasTraits, Float, CFloat
>>> class Person ( HasTraits ):
...     weight  = Float
...     cweight = CFloat
>>> bill = Person()
>>> bill.weight  = 180    # OK, 整数和浮点数匹配(转换为浮点数而不丢失信息
>>> bill.cweight = 180    # OK,
>>> bill.weight  = '180'  # Error, 字符串和浮点数不匹配
>>> bill.cweight = '180'  # OK, 调用float('180')转换为浮点
>>> print bill.cweight
180.0
```

其它类型

除了简单类型以外，Traits库还定义了许多其他的常用的数据类型。几乎所有的Trait类型都可以直接使用其名称或者当作函数调用，并且可以接受多种参数。

- **Any** : 任何对象；

  ```
  Array( [dtype = None, shape = None, value = None, typecode = No
  ```

- **Button** : 按钮类型，通常用来触发事件，参数都是用来描述界面中的按钮的样式；

  ```
  Callable( [value = None, **metadata] )
  ```

  - **CArray** : 可自动转换类型的numpy数组；调用的参数和Array相同
  - **Class** : Python老式类；

  ```
  Code( [value = "", minlen = 0, maxlen = sys.maxint, regex = "",
  ```

  - **Color** : 界面库中所采用的颜色对象；

  ```
  CSet( [trait = None, value = None, items = True, **metadata] )
  ```

  - **Constant** : 常量对象，其值不能改变，必须指定初始值；

  ```
  Dict( [key_trait = None, value_trait = None, value = None, item
  ```

- **Directory** : 表示某个目录的路径的字符串；

```
Either( val1*[, *val2, ..., valN, **metadata] )
```

- **Enum** : 枚举数据，其值可以是候选值中的一个；

```
Event( [trait = None, **metadata] )
```

- **Expression** : Python的表达式对象；

```
File( [value = "", filter = None, auto_set = False, entries = 1
```



- **Font** : 界面库中表示字体的对象；

```
class Employee(HasTraits):
    manager = self
```

定义了一个Employee类，它有一个manager属性，其类型为Employee，缺省值为对象本身：

```
>>> e = Employee()
>>> e.manager
<__main__.Employee object at 0x05DB72A0>
>>> e
<__main__.Employee object at 0x05DB72A0>
```

如果用This定义的话，那么缺省值为None。

一般来说，属性为某个类的实例时可以这样定义：

```
manager = Instance(Empolyee)
```

但是对于这个例子中，在定义manager属性时，Empolyee还不存在，因此无法如此定义。如果你喜欢这种方式的话，可以用Instance("Empolyee")来定义，当两个类的属性交叉引用时，可以使用这种字符串的方式来定义。

This和self不但可以表示类本身，还可以表示派生的类：

```
>>> from enthought.traits.api import HasTraits, This
>>> class Employee(HasTraits):
...     manager = This
...
>>> class Executive(Employee):
...   pass
...
>>> fred = Employee()
>>> mary = Executive()
>>> fred.manager = mary
>>> mary.manager = fred
```

列出可能的值

使用Enum可以定义枚举类型，在Enum的定义中给出所有可能的值，这些值必须是Python的简单数据类型，例如字符串、整数、浮点数等等，各个可能的值的类型可以不一样。可以直接将可能的值作为参数，或者将其包在某个list中，第一个值为缺省值：

```
class Items(HasTraits):
    count = Enum(None, 0, 1, 2, 3, "many")
    # 或者：
    # count = Enum([None, 0, 1, 2, 3, "many"])
```

下面是运行结果：

```
>>> item = Items()
>>> item.count = 2
>>> item.count = "many"
>>> item.count = 5
```

如果你希望候选值是可以变化的话，可以用values关键字指定定义候选值的属性名：

```
class Items(HasTraits):
    count_list = List([None, 0, 1, 2, 3, "many"])
    count = Enum(values="count_list")
```

我们定义一个count_list列表，然后在Enum定义中用values关键字指定候选值为count_list属性。

```
>>> item = Items()
>>> item.count = 5
Traceback (most recent call last)
#... 略去错误提示，此错误提示无法显示侯选值列表
>>> item.count_list.append(5)
>>> item.count = 5
>>> item.count
5
```

Trait的元数据

Trait对象可以有元数据属性，这些属性保存在HasTraits对象的trait字典中，为了解释什么是trait字典和元数据，让我们先来看一个例子：

```
from enthought.traits.api import *

class MetadataTest(HasTraits):
    i = Int(99)
    s = Str("test", desc="a string trait property")

test = MetadataTest()
```

在IPython中运行了上面的程序之后，我们对test进行如下操作：

```
>>> test.traits()
{'i': <enthought.traits.traits.CTrait object at 0x05D44EA0>
'trait_added': <enthought.traits.traits.CTrait object at 0x
's': <enthought.traits.traits.CTrait object at 0x05D44EF8>,
'trait_modified': <enthought.traits.traits.CTrait object at
```

```
>>> test.trait("i")
<enthought.traits.traits.CTrait object at 0x05D44EA0>
```

```
>>> test.trait("s").desc
'a string trait property'
```

通过调用HasTraits对象的traits方法可以得到一个包含其所有trait对象的字典。请注意，trait属性和trait对象是两个东西：

- **trait**属性：用于保存实际的值，例如：test.i, test.s
- **trait**对象：用于描述trait属性，例如：test.trait("i"), test.trait("s")

也就是说对于每一个trait属性都有一个与之对应的trait对象描述它。而元数据就是保存在trait对象中的额外的描述属性用的数据。我们看到test的trait对象除了i和s之外，还有trait_added和trait_modified，着两个在HasTraits类中定义。

元数据可以分为三类：

- 内部属性：这些属性是trait对象自带的，只读不能写
- 识别属性：这些属性是可以自由地设置的，它们可以改变trait的一些行为
- 任意属性：用户自己添加的属性，需要自己编写程序使用它们

内部元数据

下面的这些元数据属性在Traits库内部使用，用户可以读取它们的值。

- **array**：是否是数组，不是数组的trait对象没有此属性
- **default**：对应的trait属性的缺省值。也就是说：trait属性的缺省值是保存在与其对应的trait对象的元数据属性default中的：

```
>>> test.trait("i").default
99
```

- **default_kind**：一个描述缺省值的类型的字符串，其值可以是 value, list, dict, self, factory, method等：

```
>>> test.trait("i").default_kind
'value'
```

- **inner_traits**：内部的trait对象，在List, Dict等中使用，表示List和Dict内部对象的类型
- **trait_type**：描述trait属性的数据类型的对象。下面的例子中，得到的就是定义trait属性时所用的Int类的对象：

```
>>> test.trait("i").trait_type
<enthought.traits.trait_types.Int object at 0x05DBD2D0>
```

- **type**：trait属性的分类，可以是constant, delegate, event, property, trait

```
>>> test.trait("i").type
'trait'
```

能识别的元数据

下面的元数据属性不是预定义的，但是可以被HasTraits对象使用：

- **desc** : 描述trait属性用的字符串，在界面中使用
- **editor** : 指定一个生成界面时用何种TraitEditor编辑对应的trait属性
- **label** : 界面中的trait属性编辑器的标签中的字符串
- **rich_compare** : 指定判断trait属性值发生变化的方式。True(缺省)表示按值比较；Flase表示按照对象指针比较
- **trait_value** : 指定trait属性是否接受TraitValu类的对象，缺省值为False。当它为True时，将trait属性设置为TraitValue()，将重置trait属性值为缺省值。
- **transient** : 指定当对象被保存(持久化)时是否保存此trait属性值。对于大多数trait属性来说，它的缺省值都是True。

Trait事件处理

当物体的某个属性的值发生变化的时候，程序中的其它的部分可能需要响应这个变化。这种事件驱动的技术在界面程序的编写中非常常见，而Traits库给trait属性提供了事件处理功能，让我们能将事件驱动模型运用到更广泛的场景中去。

我们可以使用如下多种方法让程序监听trait属性值的变化：

- 静态命名：通过编写特定名称的函数处理trait属性值变化
- 静态修饰：用修饰函数 @on_trait_change
- 动态监听：调用on_trait_change()或者on_trait_event()将trait属性(事件源)和处理函数联系起来

静态命名的事件处理

设计自己的Trait编辑器

在前面的章节中我们知道，每种trait属性都对应有缺省的trait编辑器，如果在View中不指定编辑器的话，将使用缺省的编辑器构成界面。每个编辑器都可以对应有多个后台，目前支持的后台界面库有pyQt和wxPython。每种编辑器都可以有四种样式：simple, custom, text, readonly。

traitsUI为我们提供了很丰富的编辑器库，以至于我们很少有自己设计编辑器的需求，然而如果我们能方便地设计自己的编辑器，将能制作出更加专业的程序界面。

本章节将简要介绍trait编辑器的工作原理；并且制作一个新的trait编辑器，用以显示matplotlib提供的绘图控件；然后以此控件制作一个通用的绘制CSV文件数据图像的小工具。

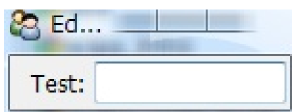
Trait编辑器的工作原理

我们先来看下面这个小程序，它定义了一个TestStrEditor类，其中有一个名为test的trait属性，其类型为Str，在view中用Item定义要在界面中显示test属性，但是没有指定它所使用的编辑器(通过editor参数)。当执行t.configure_traits()时，traits库将自动为我们挑选文本编辑框控件作为test属性的编辑器：

```
from enthought.traits.api import *
from enthought.traits.ui.api import *

class TestStrEditor(HasTraits):
    test = Str
    view = View(Item("test"))

t = TestStrEditor()
t.configure_traits()
```



使用文本编辑框控件编辑test属性

Traits库的路径

下面的介绍需要查看traits库的源程序，因此首先你需要知道它们在哪里：

traits: site-packages\Traits-3.2.0-py2.6-win32.egg\enthought\traits, 以下简称%traits%

traitsUI: site-packages\Traits-3.2.0-py2.6-win32.egg\enthought\traits\UI, 以下简称%ui%

wx后台界面库: site-packages\TraitsBackendWX-3.2.0-py2.6.egg\enthought\traitsui\wx, 以下简称 %wx%

Str对象的缺省编辑器通过其create_editor方法获得：

```
>>> from enthought.traits.api import *
>>> s = Str()
>>> ed = s.create_editor()
>>> type(ed)
<class 'enthought.traits.ui.editors.text_editor.ToolkitEditorFactor
>>> ed.get()
{'auto_set': True,
 'custom_editor_class': <class 'enthought.traits.ui.wx.text_editor.
 'enabled': True,
 'enter_set': False,
 'evaluate': <enthought.traits.ui.editors.text_editor._Identity obj
 'evaluate_name': '',
 'format_func': None,
 'format_str': '',
 'invalid': '',
 'is_grid_cell': False,
 'mapping': {},
 'multi_line': True,
 'password': False,
 'readonly_editor_class': <class 'enthought.traits.ui.wx.text_edito
 'simple_editor_class': <class 'enthought.traits.ui.wx.text_editor.
 'text_editor_class': <class 'enthought.traits.ui.wx.text_editor.Si
 'view': None}
```

create_editor方法的源代码可以在%traits%trait_types.py中的BaseStr类的定义中找到。create_editor方法得到的是一个text_editor.ToolkitEditorFactory类：

```
enthought.traits.ui.editors.text_editor.ToolkitEditorFactory
```

在%ui%editorstext_editor.py中你可以找到它的定义，它继承于EditorFactory类。EditorFactory类的代码在%ui%editor_factory.py中。EditorFactory类是Traits编辑器的核心，通过它和后台界面库联系起来。让我们来详细看看EditorFactory类中关于控件生成方面的代码：

```
class EditorFactory ( HasPrivateTraits ):
    # 下面四个属性描述四个类型的编辑器的类
    simple_editor_class = Property
    custom_editor_class = Property
    text_editor_class   = Property
    readonly_editor_class = Property

    # 用simple_editor_class创建实际的控件
    def simple_editor ( self, ui, object, name, description, parent
        return self.simple_editor_class( parent,
                                         factory     = self,
                                         ui          = ui,
                                         object      = object,
                                         name        = name,
                                         description = description

    # 这是类的方法，它通过类的以及父类自动找到与其匹配的后台界面库中的控件类
    @classmethod
    def _get_toolkit_editor(cls, class_name):
        editor_factory_classes = [factory_class for factory_class i
                                  if issubclass(factory_class, Edit
        for index in range(len( editor_factory_classes )):
            try:
                factory_class = editor_factory_classes[index]
                editor_file_name = os.path.basename(
                                sys.modules[factory_class.__module_
                return toolkit_object(':'.join([editor_file_name.sp
                                                class_name]), True)
            except Exception, e:
                if index == len(editor_factory_classes)-1:
                    raise e
        return None

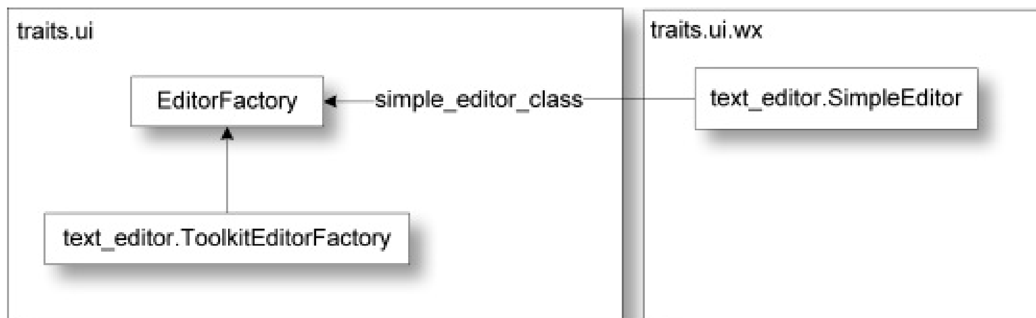
    # simple_editor_class属性的get方法，获取属性值
    def _get_simple_editor_class(self):
        try:
            SimpleEditor = self._get_toolkit_editor('SimpleEditor')
        except:
            SimpleEditor = toolkit_object('editor_factory:SimpleEdi
        return SimpleEditor
```

EditorFactory的对象有四个属性保存后台编辑器控件的类：simple_editor_class, custom_editor_class, text_editor_class, readonly_editor_class。例如前面例子中的ed对象的simple_editor_class为<class 'enthought.traits.ui.wx.text_editor.SimpleEditor'>，我们看到它用的是wx后台界面库中的text_editor中的SimpleEditor类，稍后我们将看看其内容。

EditorFactory是通过其类方法_get_toolkit_editor计算出所要用后台界面库中的类的。由于_get_toolkit_editor是类方法，它的第一个参数cls就是类本身。当调用text_editor.ToolkitEditorFactory._get_toolkit_editor()时，cls就是

text_editor.ToolkitEditorFactory类。通过调用cls.mro获得cls以及其所有父类，然后一个一个地查找，从后台界面库中找到与之匹配的类，这个工作由toolkit_object函数完成。其源代码可以在%ui%toolkit.py中找到。

因为后台界面库中的类的组织结构和traits.ui是一样的，因此不需要额外的配置文件，只需要几个字符串替代操作就可以将traits.ui中的EditorFactory类和后台界面库中的实际的编辑器类联系起来。下图显示了traits.ui中的EditorFactory和后台界面库的关系。



traits.ui中的EditorFactory和后台界面库的关系

wx后台界面库中定义了所有编辑器控件，在 %wx%text_editor.py 中你可以找到产生文本框控件的类 text_editor.SimpleEditor。类名表示了控件的样式：simple, custom, text, readonly，而其文件名(模块名)则表示了控件的类型。下面是text_editor.SimpleEditor的部分代码：

```
class SimpleEditor ( Editor ):

    # Flag for window styles:
    base_style = 0

    # Background color when input is OK:
    ok_color = OKColor

    # Function used to evaluate textual user input:
    evaluate = evaluate_trait

    def init ( self, parent ):
        """ Finishes initializing the editor by creating the underl
 widget.
 """
        factory       = self.factory
        style         = self.base_style
        self.evaluate = factory.evaluate
        self.sync_value( factory.evaluate_name, 'evaluate', 'from'

        if (not factory.multi_line) or factory.password:
            style &= ~wx.TE_MULTILINE

        if factory.password:
            style |= wx.TE_PASSWORD

        multi_line = ((style & wx.TE_MULTILINE) != 0)
        if multi_line:
            self.scrollable = True

        if factory.enter_set and (not multi_line):
            control = wx.TextCtrl( parent, -1, self.str_value,
                                   style = style | wx.TE_PROCESS_EN
            wx.EVT_TEXT_ENTER( parent, control.GetId(), self.update
        else:
            control = wx.TextCtrl( parent, -1, self.str_value, styl

        wx.EVT_KILL_FOCUS( control, self.update_object )

        if factory.auto_set:
            wx.EVT_TEXT( parent, control.GetId(), self.update_objec

        self.control = control
        self.set_tooltip()
```

真正产生控件的程序是在init方法中，此方法在产生界面时自动被调用，注意方法名是init，不要和对象初始化方法__init__搞混淆了。

制作matplotlib的编辑器

Enthought的官方绘图库是采用Chaco，不过如果你对matplotlib库更加熟悉的话，将matplotlib的绘图控件嵌入TraitsUI界面中将是非常有用的。下面先来看一下嵌入matplotlib控件的完整源代码：

```
# -*- coding: utf-8 -*-
# file name: mpl_figure_editor.py
import wx
import matplotlib
# matplotlib采用WXAgg为后台，这样才能将绘图控件嵌入以wx为后台界面库的traits
matplotlib.use("WXAgg")
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as
from matplotlib.backends.backend_wx import NavigationToolbar2Wx
from enthought.traits.ui.wx.editor import Editor
from enthought.traits.ui.basic_editor_factory import BasicEditorFac

class _MPLFigureEditor(Editor):
    """
 相当于wx后台界面库中的编辑器，它负责创建真正的控件
 """
    scrollable = True

    def init(self, parent):
        self.control = self._create_canvas(parent)
        self.set_tooltip()
        print dir(self.item)

    def update_editor(self):
        pass

    def _create_canvas(self, parent):
        """
 创建一个Panel, 布局采用垂直排列的BoxSizer, panel中中添加
 FigureCanvas, NavigationToolbar2Wx, StaticText三个控件
 FigureCanvas的鼠标移动事件调用mousemoved函数，在StaticText
 显示鼠标所在的数据坐标
 """
        panel = wx.Panel(parent, -1, style=wx.CLIP_CHILDREN)
        def mousemoved(event):
            panel.info.SetLabel("%s, %s" % (event.xdata, event.ydat
        panel.mousemoved = mousemoved
        sizer = wx.BoxSizer(wx.VERTICAL)
        panel.SetSizer(sizer)
        mpl_control = FigureCanvas(panel, -1, self.value)
        mpl_control.mpl_connect("motion_notify_event", mousemoved)
        toolbar = NavigationToolbar2Wx(mpl_control)
        sizer.Add(mpl_control, 1, wx.LEFT | wx.TOP | wx.GROW)
        sizer.Add(toolbar, 0, wx.EXPAND|wx.RIGHT)
        panel.info = wx.StaticText(parent, -1)
        sizer.Add(panel.info)

        self.value.canvas.SetMinSize((10,10))
        return panel
```

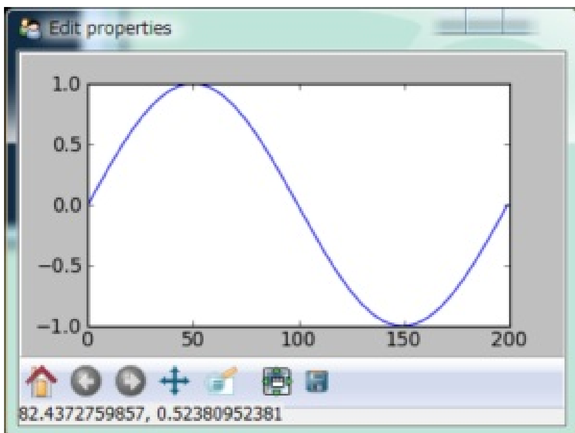
```
class MPLFigureEditor(BasicEditorFactory):
    """
 相当于traits.ui中的EditorFactory，它返回真正创建控件的类
 """
    klass = _MPLFigureEditor

if __name__ == "__main__":
    from matplotlib.figure import Figure
    from enthought.traits.api import HasTraits, Instance
    from enthought.traits.ui.api import View, Item
    from numpy import sin, cos, linspace, pi

    class Test(HasTraits):
        figure = Instance(Figure, ())
        view = View(
            Item("figure", editor=MPLFigureEditor(), show_label=Fal
            width = 400,
            height = 300,
            resizable = True)
        def __init__(self):
            super(Test, self).__init__()
            axes = self.figure.add_subplot(111)
            t = linspace(0, 2*pi, 200)
            axes.plot(sin(t))

    Test().configure_traits()
```

此程序的运行结果如下：



在TraitsUI界面中嵌入的matplotlib绘图控件

由于我们的编辑器没有simple等四种样式，也不会放到wx后台界面库的模块中，因此不能采用上节所介绍的自动查找编辑器类的办法。traits.ui为我们提供一个一个方便的类来完成这些操作：BasicEditorFactory。它的源程序可以在%ui%basic_editor_factory.py中找到。下面是其中的一部分：

```
class BasicEditorFactory ( EditorFactory ):
    klass = Any

    def _get_simple_editor_class ( self ):
        return self.klass
    ...
```

它通过重载EditorFactory中的simple_editor_class属性，直接返回创建控件的库klass。MPLFigureEditor继承于BasicEditorFactory，指定创建控件的类为_MPLFigureEditor。

和text_editor.SimpleEditor一样，从Editor类继承，在_MPLFigureEditor类的init方法中，创建实际的控件。因为Editor类中有一个update_editor方法，在其对应的trait属性改变是会被调用，而我们的绘图控件不需要这个功能，所以重载update_editor，让它不做任何事情。

matplotlib中，在创建FigureCanvas时需要指定与其对应的Figure对象：

```
mpl_control = FigureCanvas(panel, -1, self.value)
```

这里self.value就是这个Figure对象，它在MVC的模型类Test中被定义为：

```
figure = Instance(Figure, ())
```

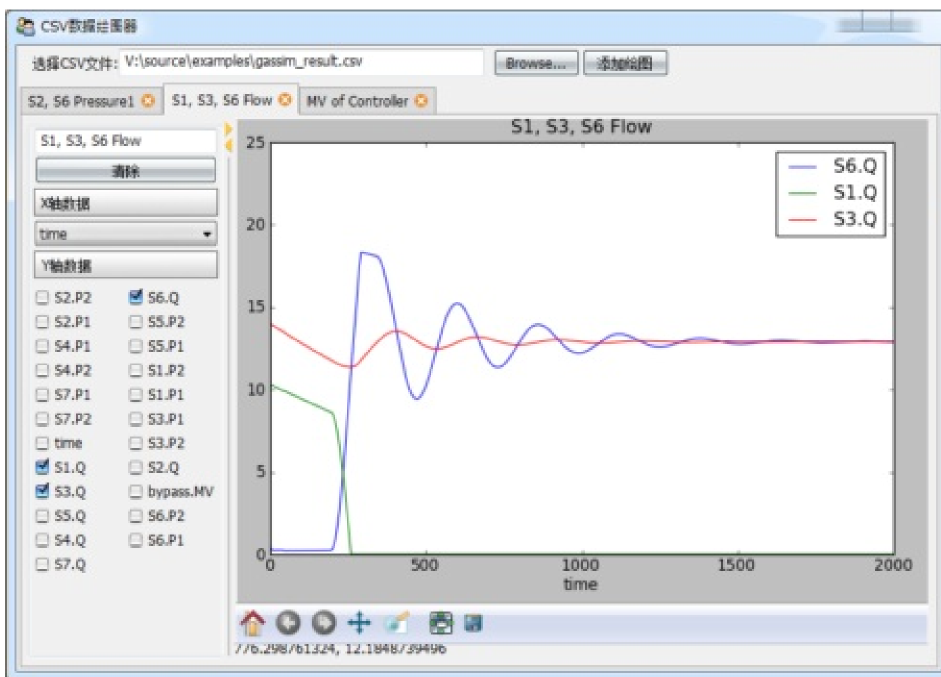
控件类可以通过self.value获得与其对应的模型类中的对象。因此_MPLFigureEditor中的self.value和Test类中的self.figure是同一个对象。

_create_canvas方法中的程序编写和在一个标准的wx窗口中添加控件是一样的，界面库相关的细节不是本书的重点，因此不再详细解释了。读者可以参照matplotlib和wxPython的相应文档。

CSV数据绘图工具

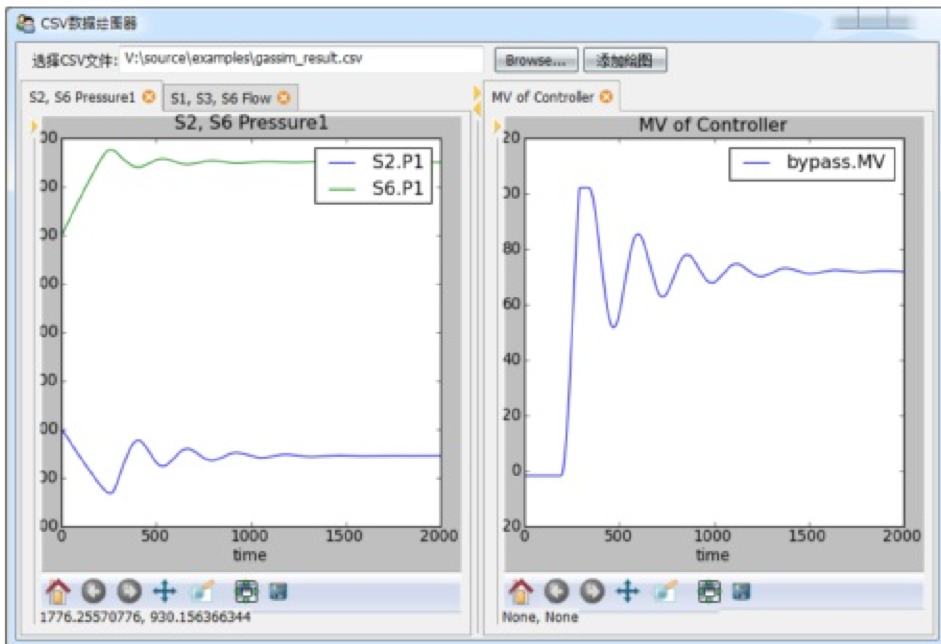
下面用前面介绍的matplotlib编辑器制作一个CSV数据绘图工具。用此工具打开一个CSV数据文档之后，可以绘制多个X-Y坐标图。用户可以自由地添加新的坐标图，修改坐标图的标题，选择坐标图的X轴和Y轴的数据。

下面是此程序的界面截图：



CSV数据绘图工具的界面

图中以标签页的形式显示多个绘图，用户可以从左侧的数据选择栏中选择X轴和Y轴的数据。标签页可以自由的拖动，构成上下左右分栏，并且可以隐藏左侧的数据选择栏：



使用可调整DOCK的多标签页界面方便用户对比数据

由于绘图控件是matplotlib所提供的，因此平移、缩放、保存文件等功能也一应俱全。由于所有的界面都是采用TraitsUI设计的，因此主窗口既可以用来单独显示，也可以嵌入到一个更大的界面中，运用十分灵活。

下面是完整的源程序，运行时需要和mpl_figure_editor.py放在一个文件夹下。包括注释程序一共约170行，编写时间少于一小时。

```
# -*- coding: utf-8 -*-
from matplotlib.figure import Figure
from mpl_figure_editor import MPLFigureEditor
from enthought.traits.ui.api import *
from enthought.traits.api import *
import csv

class DataSource(HasTraits):
    """
 数据源，data是一个字典，将字符串映射到列表
 names是data中的所有字符串的列表
 """
    data = DictStrAny
    names = List(Str)

    def load_csv(self, filename):
        """
 从CSV文件读入数据，更新data和names属性
 """
        f = file(filename)
        reader = csv.DictReader(f)
        self.names = reader.fieldnames
        for field in reader.fieldnames:
            self.data[field] = []
        for line in reader:
            for k, v in line.iteritems():
                self.data[k].append(float(v))
        f.close()

class Graph(HasTraits):
    """
 绘图组件，包括左边的数据选择控件和右边的绘图控件
 """
    name = Str # 绘图名，显示在标签页标题和绘图标题中
    data_source = Instance(DataSource) # 保存数据的数据源
    figure = Instance(Figure) # 控制绘图控件的Figure对象
    selected_xaxis = Str # X轴所用的数据名
    selected_items = List # Y轴所用的数据列表

    clear_button = Button(u"清除") # 快速清除Y轴的所有选择的数据

    view = View(
        HSplit( # HSplit分为左右两个区域，中间有可调节宽度比例的调节手柄
            # 左边为一个组
            VGroup(
                Item("name"),   # 绘图名编辑框
                Item("clear_button"), # 清除按钮
                Heading(u"X轴数据"),  # 静态文本
                # X轴选择器，用EnumEditor编辑器，即ComboBox控件，控件中的
                # data_source的names属性得到
                Item("selected_xaxis", editor=
                    EnumEditor(name="object.data_source.names", fo
```

```
                Heading(u"Y轴数据"), # 静态文本
                # Y轴选择器，由于Y轴可以多选，因此用CheckBox列表编辑，按两
                Item("selected_items", style="custom",
                     editor=CheckListEditor(name="object.data_sourc
                            cols=2, format_str=u"%s")),
                show_border = True, # 显示组的边框
                scrollable = True,  # 组中的控件过多时，采用滚动条
                show_labels = False # 组中的所有控件都不显示标签
            ),
            # 右边绘图控件
            Item("figure", editor=MPLFigureEditor(), show_label=Fal
        )
    )

    def _name_changed(self):
        """
当绘图名发生变化时，更新绘图的标题
"""
        axe = self.figure.axes[0]
        axe.set_title(self.name)
        self.figure.canvas.draw()

    def _clear_button_fired(self):
        """
清除按钮的事件处理
"""
        self.selected_items = []
        self.update()

    def _figure_default(self):
        """
figure属性的缺省值，直接创建一个Figure对象
"""
        figure = Figure()
        figure.add_axes([0.05, 0.1, 0.9, 0.85]) #添加绘图区域，四周留有
        return figure

    def _selected_items_changed(self):
        """
Y轴数据选择更新
"""
        self.update()

    def _selected_xaxis_changed(self):
        """
X轴数据选择更新
"""
        self.update()

    def update(self):
        """
重新绘制所有的曲线
"""
```

```
        axe = self.figure.axes[0]
        axe.clear()
        try:
            xdata = self.data_source.data[self.selected_xaxis]
        except:
            return
        for field in self.selected_items:
            axe.plot(xdata, self.data_source.data[field], label=fie
        axe.set_xlabel(self.selected_xaxis)
        axe.set_title(self.name)
        axe.legend()
        self.figure.canvas.draw()

class CSVGrapher(HasTraits):
    """
 主界面包括绘图列表，数据源，文件选择器和添加绘图按钮
 """
    graph_list = List(Instance(Graph)) # 绘图列表
    data_source = Instance(DataSource) # 数据源
    csv_file_name = File(filter=[u"*.csv"]) # 文件选择
    add_graph_button = Button(u"添加绘图") # 添加绘图按钮

    view = View(
        # 整个窗口分为上下两个部分
        VGroup(
            # 上部分横向放置控件，因此用HGroup
            HGroup(
                # 文件选择控件
                Item("csv_file_name", label=u"选择CSV文件", width=40
                # 添加绘图按钮
                Item("add_graph_button", show_label=False)
            ),
            # 下部分是绘图列表，采用ListEditor编辑器显示
            Item("graph_list", style="custom", show_label=False,
                 editor=ListEditor(
                     use_notebook=True, # 是用多标签页格式显示
                     deletable=True, # 可以删除标签页
                     dock_style="tab", # 标签dock样式
                     page_name=".name") # 标题页的文本使用Graph对象的n
                )
        ),
        resizable = True,
        height = 0.8,
        width = 0.8,
        title = u"CSV数据绘图器"
    )

    def _csv_file_name_changed(self):
        """
 打开新文件时的处理，根据文件创建一个DataSource
 """
        self.data_source = DataSource()
        self.data_source.load_csv(self.csv_file_name)
```

```
        del self.graph_list[:]

    def _add_graph_button_changed(self):
        """
 添加绘图按钮的事件处理
 """
        if self.data_source != None:
            self.graph_list.append( Graph(data_source = self.data_s

if __name__ == "__main__":
    csv_grapher = CSVGrapher()
    csv_grapher.configure_traits()
```

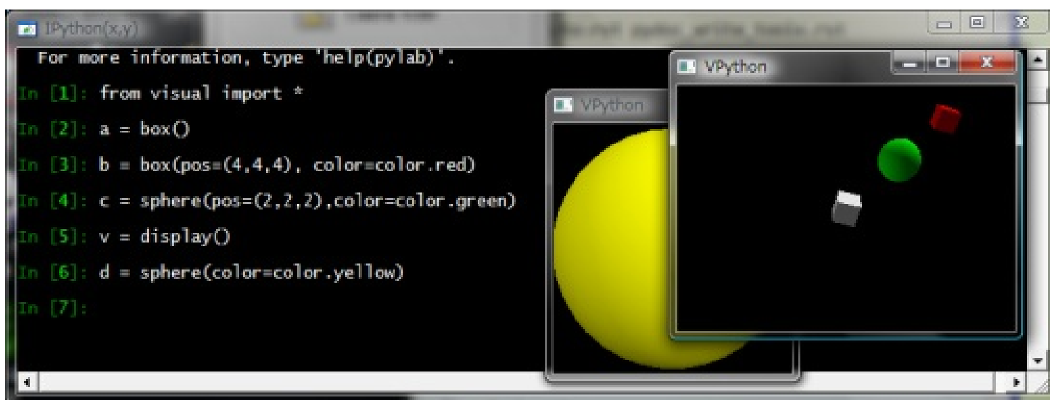
程序中已经有比较详细的注释，这里就不再重复。如果你对traits库的某项用法还不太了解的话，可以直接查看其源代码，代码中都有详细的注释。下面是几个比较重点的部分：

- 整个程序的界面处理都只是组装View对象，看不到任何关于控件操作的代码，因此大大地节省了程序的开发时间。
- 通过配置141行的ListEditor，使其用标签页的方式显示graph_list中的每个元素，以此管理多个Graph对象。
- 在43行中，Graph类用HSplit将其数据选择部分和绘图控件部分分开，HSplit提供的更改左右部分的比例和隐藏的功能。
- 本书写作时所采用的traitsUI库版本为3.2，如果在标签页标题中输入中文，会出现错误，这是因为TraitsUI中还有些代码对unicode的支持不够，希望日后会有所改善。目前可以通过分析错误提示信息，修改TraitsUI库的源代码，只需要将下面提示中的770行中的str改为unicode既可以修复。

> ```
> >>> from visual import *
> ```

之后就可以随心所欲的调用visual库通过的函数。需要注意的是如果你关闭了visual弹出的场景窗口的话，ipython对话也随之结束。如果你需要关闭场景窗口可以用下面的语句：

```
>>> scene.visible = False
```



在IPython中交互式地观察visual的运行结果

上图是用IPython交互式的使用visual的一个例子，可以看到通过IPython能够控制多个场景窗口。

- 场景窗口
    - 控制场景窗口
    - 控制照相机

Visual使用手册

在ipython中交互式地使用

visual库可以在IPython中交互式的使用，启动ipython之后，只需要先执行：

```
>>> from visual import *
```

之后就可以随心所欲的调用visual库通过的函数。需要注意的是如果你关闭了visual弹出的场景窗口的话，ipython对话也随之结束。如果你需要关闭场景窗口可以用下面的语句：

```
>>> scene.visible = False
```

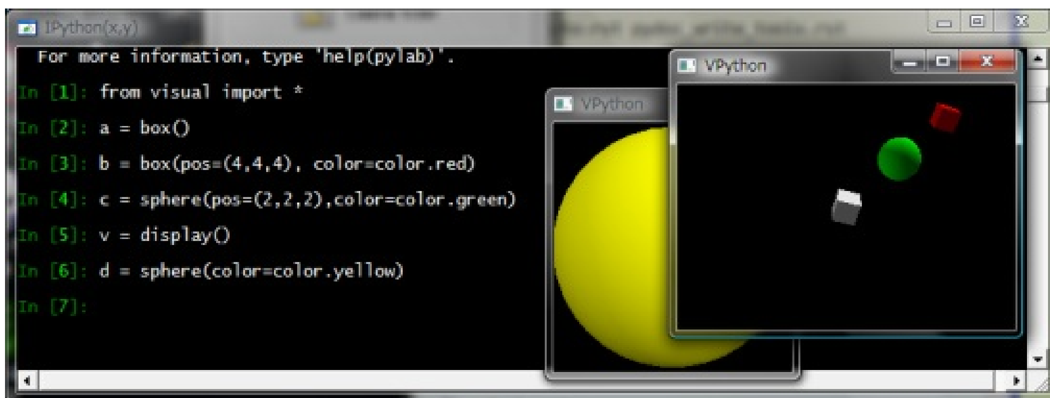


图14.1 在IPython中交互式地观察visual的运行结果

上图是用IPython交互式的使用visual的一个例子，可以看到通过IPython能够控制多个场景窗口。

场景窗口

visual中的所有的3D物体都在一个窗口中显示，此窗口为display类的对象，通过这对象我们可以修改窗口的各种属性；控制场景中的照相机，从各个角度观察场景。

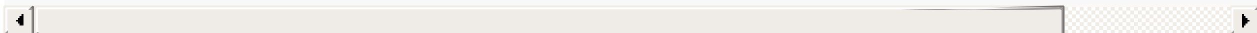
控制场景窗口

从visual库载入所有对象之后，缺省情况下，有一个可以用变量scene访问的缺省的场景窗口对象，它也是初始情况下的当前窗口：

```
>>> from visual import *
>>> scene
<visual.ui.display object at 0x032BF600>
```

我们看到场景窗口对象是visual.ui.display类的一个实例。真正的窗口需要在其中放置物体才会被显示出来。因此如果我们用display()创建自己的窗口对象的话，那么可以不用管这个缺省的窗口对象，我们创建的窗口对象将变成当前窗口。用box等类创建的3D物体将会被放到当前窗口中。下面语句调用display创建一个新的窗口对象：

```
>>> scene2 = display(title='Scene2', x=0, y=0, width=600, height=2(
 center=(5,0,0), background=(0,1,1))
```

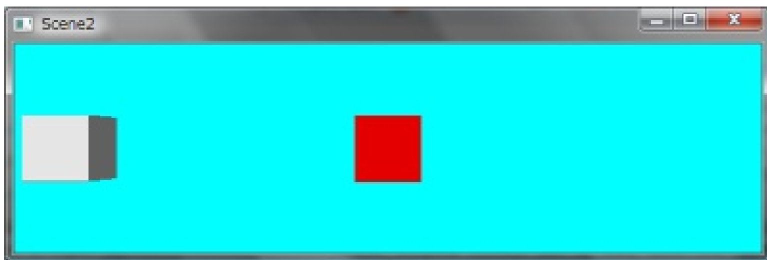


执行上面的语句之后，将创建一个标题为Scene2的窗口，其左上角的坐标为(0,0)，宽度为600像素，高度为200像素，照相机所正对的位置的坐标(5,0,0)，也就是说窗口中心的点的3D坐标为(5,0,0)，背景为青色。注意要显示窗口，我们需要往里面放物体：

```
>>> box()
>>> <visual.primitives.box object at 0x0334F090>
```

```
>>> box(pos=(5,0,0), color=color.red)
>>> <visual.primitives.box object at 0x0334F120>
```

第一个立方体放在了缺省坐标(0,0,0)处，其颜色为缺省的白色；第二个立方体放在了坐标(5,0,0)处，颜色为红色。红色立方体在窗口的中心，和我们设置的窗口的center属性一致。

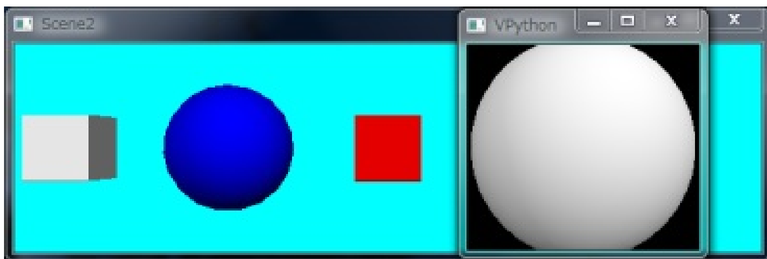


在场景中放置立方体

我们可以调用窗口对象的select方法使其成为当前窗口。通过display.get_selected()可以获得当前窗口对象：

```
>>> scene.select()
>>> sphere()
<visual.primitives.sphere object at 0x0331D7B0>
>>> scene2.select()
>>> sphere(pos=(2.5,0,0), color=color.blue)
<visual.primitives.sphere object at 0x0331D810>
>>> display.get_selected() == scene2
True
```

上面的程序先将scene改为当前窗口，然后在其中创建一个球体；接着将scene2改为当前窗口，在其中创建一个蓝色的球体，放在坐标(2.5,0,0)处。最后调用display.get_selected()检查当前窗口是否是scene2。执行这段程序之后，将出现两个场景窗口，缺省窗口的标题为VPython，其中有一个球体；我们自己创建的窗口标题为Scene2，其中有两个立方体和一个球体。



在第二个场景中放置球体

窗口对象有如下的属性：

- **foreground**：在窗口中创建物体时所采用的缺省颜色，缺省值为白色。例如运行 scene.foreground = color.green 之后，窗口中新添加的物体如果不指定颜色的话就会是绿色的。
- **background**：窗口的背景颜色，缺省值为黑色。
- **ambient**：环境光的颜色，缺省值为color.gray(0.2)，为了和visual 3兼容，使用scene.ambient=0.2和scene.ambient=color.gray(0.2)是一样的。
- **lights**：场景窗口中的光源列表，场景中的缺省光源为：

```
[distant_light(direction=(0.22, 0.44, 0.88), color=color.gr
 distant_light(direction=(-0.88, -0.22, -0.44), color=color
```

可以用如下的语句查看光源的属性：

```
&gt;&gt;&gt; scene.lights[0].direction
vector(0.218217890235992, 0.436435780471985, 0.872871560943
```

- **cursor.visible**：控制场景窗口中鼠标是否显示，如果设置为False的话，那么鼠标将被隐藏。你可以用它在用鼠标拖拽物体时隐藏鼠标，释放物体时显示鼠标。
- **objects**：窗口中所有可见的物体的列表，被隐藏的物体和光源不在此列表之中，当通过设置某物体的visible属性隐藏它时，其效果就是将它从此列表中删除。下面的语句让场景中所有的box都变成红色：

```
for obj in scene2.objects:
    if isinstance(obj, box):
        obj.color = color.red
```

- **show_rendertime**：如果其值为true，那么在窗口的左下角将显示如"cycle:27: 5" 的字样。它表示场景润色的帧之间的间隔为27毫秒，每帧需要5毫秒时间润色。这表明用户的Python程序每帧有22毫秒的处理时间。
- **stero**：立体视觉设置。如果你有双色3D立体眼镜的话，不妨试试这个选项。例如scene.stereo="redcyan"，将润色为红-青立体眼镜用的场景，此外还有"redblue"和"yellowblue"等选项。此外还可以设置为"crosseyed"，它将润色左右两个场景，当你左右眼交叉聚焦到右左两个图时，产生立体效果。和流行一时的立体图片类似，反正我是看不出来。设置为"active"的话，产生可以用shutter glasses观看的立体场景。
- **stereodepth**：修改立体视觉的深度，缺省值为0，设置为2有最好的立体效果。这个参数我没有用过。

下面的属性 x, y, width, height, title和fullscreen等都只能在窗口隐藏的时候修改。因此通常是在用display创建窗口的时候同时设置这些属性。如果你需要设置已经显示了的窗口的属性的话，先通过设置visible = False将其隐藏，设置这些属性，最后再重新显示窗口。

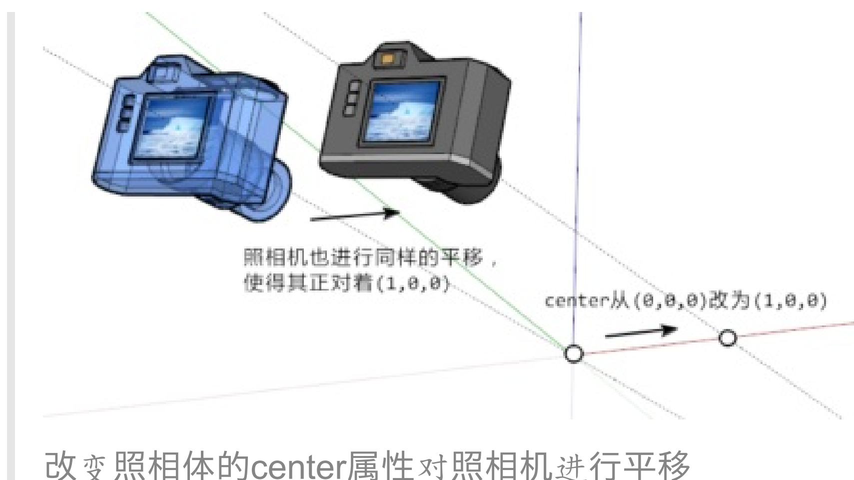
- **x, y**：窗口在屏幕中的位置，窗口左上角的屏幕坐标
- **width, height**：整个窗口的像素宽度和高度，包括边框和标题栏
- **title**：窗口的标题栏中的文字，如果需要设置中文的话，需要设置为windows的缺省编码，例如scene.title = u"中文标题".encode("gb2312")
- **fullscreen**：全屏显示，如果用scene.fullscreen=True全屏显示的话，将没有窗口的边框和标题栏，按Esc键退出

- **visible** : 窗口是否可见，当某个物体被添加进窗口时，窗口将自动的被设置为可见
- **exit** : 当exit为False的时候，将禁止窗口的关闭按钮，也就是无法通过关闭按钮关闭窗口，缺省值为True。

控制照相机

照相机的控制是通过设置窗口的属性来完成的。

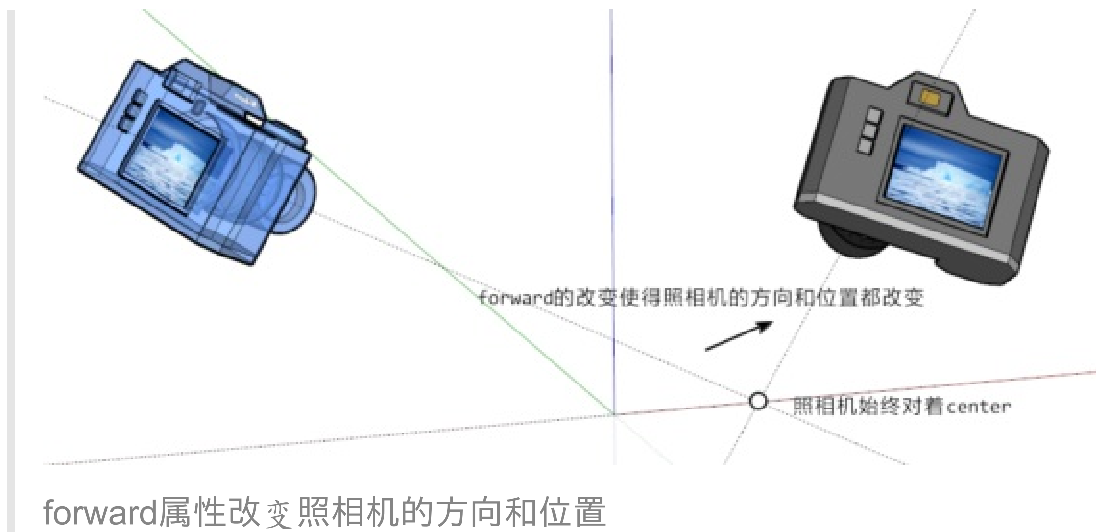
- **center** : 照相机所正对的3D空间的坐标点，即使用户旋转场景，照相机也始终正对着这个坐标。如果你修改了center的值的话，照相机将保持其方向不变，进行平行移动使得其正对center坐标。center的缺省值为(0,0,0)。



改变照相体的center属性对照相机进行平移

- **autocenter** : 如果设置为True的话，将自动计算center属性，使得它为包含所有物体的最小的长方体的中心，此最小长方体的各边与x, y, z轴平行。这样，照相机始终跟随着场景中的物体，因此如果你移动了场景中的任何物体，都有可能改变center属性。

- **forward** : 照相机所指向的方向。也就是从照相机所在的位置到center的方向矢量。用户不能直接修改照相机所在的位置，但可以通过scene.mouse.camera获得。当用户旋转场景时，其实就是在修改forward属性。当forward被修改之后，照相机将会改变其位置使得其方向和forward矢量平行，其中心正对center点。forward的缺省值为(0,0,-1)，因此是从上往下的俯视观察场景。



forward属性改变照相机的方向和位置

声音的输入输出

在本章我们将学习如何读写WAV文件，如何利用声卡实时地进行声音的输入输出。标准的Python已经支持WAV文件的读写，而实时的声音输入输出需要安装pyAudio(http://people.csail.mit.edu/hubert/pyaudio)。最后我们还将看看如何使用pyMedia(http://pymedia.org)进行Mp3的解码和播放。

掌握了上面的基础知识之后，就可以做许多有趣的声效处理的算法实验了。声效处理方面的内容将在以后的章节详细介绍。

读写Wave文件

WAV是Microsoft开发的一种声音文件格式，虽然它支持多种压缩格式，不过它通常被用来保存未压缩的声音数据（PCM脉冲编码调制)。WAV有三个重要的参数：声道数、取样频率和量化位数。

- 声道数：可以是单声道或者是双声道
- 采样频率：一秒内对声音信号的采集次数，常用的有8kHz, 16kHz, 32kHz, 48kHz, 11.025kHz, 22.05kHz, 44.1kHz
- 量化位数：用多少bit表达一次采样所采集的数据，通常有8bit、16bit、24bit和32bit等几种

例如CD中所储存的声音信号是双声道、44.1kHz、16bit。

如果你需要自己录制和编辑声音文件，推荐使用Audacity(http://audacity.sourceforge.net)，它是一款开源的、跨平台、多声道的录音编辑软件。在我的工作中经常使用Audacity进行声音信号的录制，然后再输出成WAV文件供Python程序处理。

读Wave文件

下面让我们来看看如何在Python中读写声音文件：

```
# -*- coding: utf-8 -*-
import wave
import pylab as pl
import numpy as np

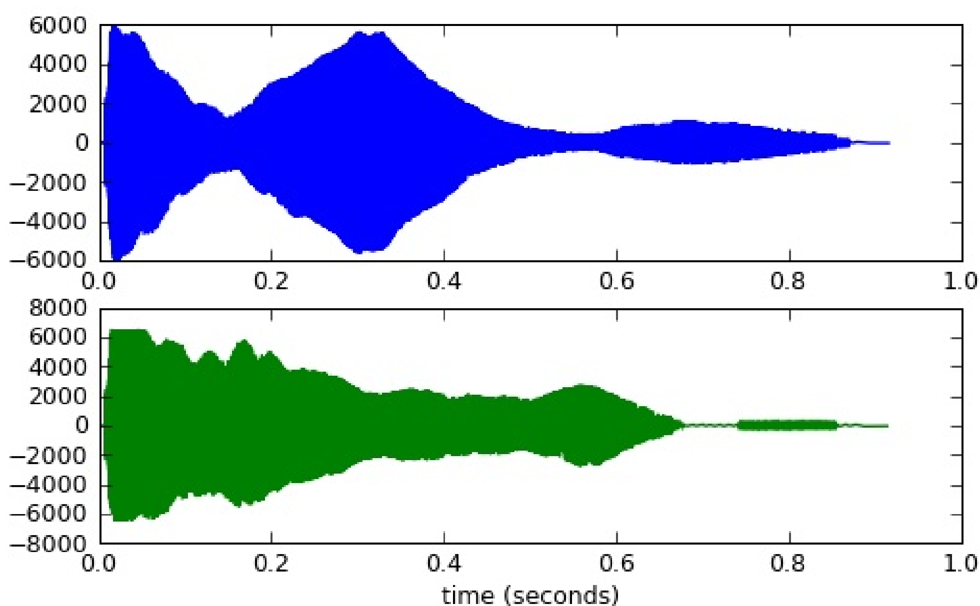
# 打开WAV文档
f = wave.open(r"c:\WINDOWS\Media\ding.wav", "rb")

# 读取格式信息
# (nchannels, sampwidth, framerate, nframes, comptype, compname)
params = f.getparams()
nchannels, sampwidth, framerate, nframes = params[:4]

# 读取波形数据
str_data = f.readframes(nframes)
f.close()

#将波形数据转换为数组
wave_data = np.fromstring(str_data, dtype=np.short)
wave_data.shape = -1, 2
wave_data = wave_data.T
time = np.arange(0, nframes) * (1.0 / framerate)

# 绘制波形
pl.subplot(211)
pl.plot(time, wave_data[0])
pl.subplot(212)
pl.plot(time, wave_data[1], c="g")
pl.xlabel("time (seconds)")
pl.show()
```



WindowsXP的经典"叮"声的波形

首先载入Python的标准处理WAV文件的模块，然后调用wave.open打开wav文件，注意需要使用"rb"(二进制模式)打开文件：

```
import wave
f = wave.open(r"c:\WINDOWS\Media\ding.wav", "rb")
```

open返回一个的是一个Wave_read类的实例，通过调用它的方法读取WAV文件的格式和数据：

- getparams：一次性返回所有的WAV文件的格式信息，它返回的是一个组元(tuple)：声道数, 量化位数（byte单位）, 采样频率, 采样点数, 压缩类型, 压缩类型的描述。wave模块只支持非压缩的数据，因此可以忽略最后两个信息：

  ```
  params = f.getparams()
  nchannels, sampwidth, framerate, nframes = params[:4]
  ```

- getnchannels, getsampwidth, getframerate, getnframes等方法可以单独返回WAV文件的特定的信息。
- readframes：读取声音数据，传递一个参数指定需要读取的长度（以取样点为单位），readframes返回的是二进制数据（一大堆bytes)，在Python中用字符串表示二进制数据：

  ```
  str_data = f.readframes(nframes)
  ```

接下来需要根据声道数和量化单位，将读取的二进制数据转换为一个可以计算的数组：

```
wave_data = np.fromstring(str_data, dtype=np.short)
```

通过fromstring函数将字符串转换为数组，通过其参数dtype指定转换后的数据格式，由于我们的声音格式是以两个字节表示一个取样值，因此采用short数据类型转换。现在我们得到的wave_data是一个一维的short类型的数组，但是因为我们的声音文件是双声道的，因此它由左右两个声道的取样交替构成：LRLRLRLR....LR（L表示左声道的取样值，R表示右声道取样值）。修改wave_data的sharp之后：

```
wave_data.shape = -1, 2
```

将其转置得到：

```
wave_data = wave_data.T
```

整个转换过程如下图所示：

最后通过取样点数和取样频率计算出每个取样的时间：

```
time = np.arange(0, nframes) * (1.0 / framerate)
```

写Wave文件

写WAV文件的方法和读类似：

```
# -*- coding: utf-8 -*-
import wave
import numpy as np
import scipy.signal as signal

framerate = 44100
time = 10

# 产生10秒44.1kHz的100Hz - 1kHz的频率扫描波
t = np.arange(0, time, 1.0/framerate)
wave_data = signal.chirp(t, 100, time, 1000, method='linear') * 100
wave_data = wave_data.astype(np.short)

# 打开WAV文档
f = wave.open(r"sweep.wav", "wb")

# 配置声道数、量化位数和取样频率
f.setnchannels(1)
f.setsampwidth(2)
f.setframerate(framerate)
# 将wav_data转换为二进制数据写入文件
f.writeframes(wave_data.tostring())
f.close()
```

10-12行通过调用scipy.signal库中的chrip函数，产生长度为10秒、取样频率为44.1kHz、100Hz到1kHz的频率扫描波。由于chrip函数返回的数组为float64型，需要调用数组的astype方法将其转换为short型。

18-20行分别设置输出WAV文件的声道数、量化位数和取样频率，当然也可以调用文件对象的setparams方法一次性配置所有的参数。最后21行调用文件的writeframes方法，将数组的内部的二进制数据写入文件。writeframes方法会自动的更新WAV文件头中的长度信息(nframes)，保证其和真正的数据数量一致。

用pyAudio播放和录音

通过上一节介绍的读写声音文件的方法，我们可以离线处理已经录制好的声音。不过更酷的是我们可以通过pyAudio库从声卡读取声音数据，处理之后再写回声卡，这样就可以在电脑上实时地输入、处理和输出声音数据。想象一下，我们可以做一个小程序，读取麦克风的数据；加上回声并和WAV文件中的数据进行混合；最后从声卡输出。这不就是一个Karaoke的原型么。

pyAudio是开源声音库PortAudio(http://www.portaudio.com)的Python绑定，目前它只支持阻塞式的输入输出模式。所谓阻塞式就是需要用户的程序主动地去读写输入输出流。虽然阻塞式在功能上有所局限，但是由于编程比较简单，非常适合一些处理声音的脚本程序开发。

播放

下面先来看看如何用pyAudio播放声音。

```
# -*- coding: utf-8 -*-
import pyaudio
import wave

chunk = 1024

wf = wave.open(r"c:\WINDOWS\Media\ding.wav", 'rb')

p = pyaudio.PyAudio()

# 打开声音输出流
stream = p.open(format = p.get_format_from_width(wf.getsampwidth()
                channels = wf.getnchannels(),
                rate = wf.getframerate(),
                output = True)

# 写声音输出流进行播放
while True:
    data = wf.readframes(chunk)
    if data == "": break
    stream.write(data)

stream.close()
p.terminate()
```

这段程序首先根据WAV文件的量化格式、声道数和取样频率，分别配置open函数的各个参数，然后循环从WAV文件读取数据，写入用open函数打开的声音输出流。我们看到17-20行的while循环没有任何等待的代码。因为pyAudio使用阻塞模式，因此当底层的输出数据缓存没有空间保存数据时，stream.write会阻塞用户程序，直到stream.write能将数据写入输出缓存。

PyAudio类的open函数有许多参数：

- **rate** - 取样频率

- **channels** - 声道数
- **format** - 取样值的量化格式 (paFloat32, paInt32, paInt24, paInt16, paInt8 ...)。在上面的例子中，使用get_format_from_width方法将wf.sampwidth()的返回值2转换为paInt16
- **input** - 输入流标志，如果为True的话则开启输入流
- **output** - 输出流标志，如果为True的话则开启输出流
- **input_device_index** - 输入流所使用的设备的编号，如果不指定的话，则使用系统的缺省设备
- **output_device_index** - 输出流所使用的设备的编号，如果不指定的话，则使用系统的缺省设备
- **frames_per_buffer** - 底层的缓存的块的大小，底层的缓存由N个同样大小的块组成
- **start** - 指定是否立即开启输入输出流，缺省值为True

录音

从声卡读取数据和写入数据一样简单，下面我们用一个简单的声音监测小程序来展示一下如何用pyAudio读取声音数据。

```
# -*- coding: utf-8 -*-
from pyaudio import PyAudio, paInt16
import numpy as np
from datetime import datetime
import wave

# 将data中的数据保存到名为filename的WAV文件中
def save_wave_file(filename, data):
    wf = wave.open(filename, 'wb')
    wf.setnchannels(1)
    wf.setsampwidth(2)
    wf.setframerate(SAMPLING_RATE)
    wf.writeframes("".join(data))
    wf.close()

NUM_SAMPLES = 2000      # pyAudio内部缓存的块的大小
SAMPLING_RATE = 8000    # 取样频率
LEVEL = 1500            # 声音保存的阈值
COUNT_NUM = 20          # NUM_SAMPLES个取样之内出现COUNT_NUM个大于LEVE
SAVE_LENGTH = 8         # 声音记录的最小长度：SAVE_LENGTH * NUM_SAMPLE

# 开启声音输入
pa = PyAudio()
stream = pa.open(format=paInt16, channels=1, rate=SAMPLING_RATE, in
                frames_per_buffer=NUM_SAMPLES)

save_count = 0
save_buffer = []

while True:
    # 读入NUM_SAMPLES个取样
```

```
        string_audio_data = stream.read(NUM_SAMPLES)
        # 将读入的数据转换为数组
        audio_data = np.fromstring(string_audio_data, dtype=np.short)
        # 计算大于LEVEL的取样的个数
        large_sample_count = np.sum( audio_data > LEVEL )
        print np.max(audio_data)
        # 如果个数大于COUNT_NUM，则至少保存SAVE_LENGTH个块
        if large_sample_count > COUNT_NUM:
            save_count = SAVE_LENGTH
        else:
            save_count -= 1

        if save_count < 0:
            save_count = 0

        if save_count > 0:
            # 将要保存的数据存放到save_buffer中
            save_buffer.append( string_audio_data )
        else:
            # 将save_buffer中的数据写入WAV文件，WAV文件的文件名是保存的时刻
            if len(save_buffer) > 0:
                filename = datetime.now().strftime("%Y-%m-%d_%H_%M_%S")
                save_wave_file(filename, save_buffer)
                save_buffer = []
                print filename, "saved"
```

此程序一开头是一系列的全局变量，用来配置录音的一些参数：以SAMPLING_RATE为采样频率，每次读入一块有NUM_SAMPLES个采样的数据块，当读入的采样数据中有COUNT_NUM个值大于LEVEL的取样的时候，将数据保存进WAV文件，一旦开始保存数据，所保存的数据长度最短为SAVE_LENGTH个块。WAV文件以保存时的时刻作为文件名。

从声卡读入的数据和从WAV文件读入的类似，都是二进制数据，由于我们用paInt16格式(16bit的short类型)保存采样值，因此将它自己转换为dtype为np.short的数组。

用pyMedia播放Mp3

数字信号系统

FIR和IIR滤波器

在数字信号处理领域中，数字滤波器占有非常重要的地位。根据其计算方式可以分为FIR(有限脉冲响应)滤波器，和IIR(无限脉冲响应)滤波器两种。

FIR滤波器根据如下公式进行计算：

$$y[m] = b[0]x[m] + b[1]x[m-1] + \cdots + b[P]x[n-P]$$

IIR滤波器根据如下公式(直接1型)进行计算：

$$\begin{aligned} y[m] &= b[0]x[m] + b[1]x[m-1] + \cdots + b[P]x[m-P] \\ &- a[1]y[m-1] - a[2]y[m-2] - \cdots - a[Q]y[m-Q] \end{aligned}$$

其中x是输入信号，数组a和b是滤波器的系数，y是滤波器的输出。我们可以把FIR滤波器看作是IIR滤波器的一种特殊情况：当系数a都为0时就从IIR滤波器变为了FIR滤波器了。

根据FIR滤波器的计算公式我们可以知道，时刻m的输出y[m]由时刻m的输入x[m]以及之前的输入x[m-1] ... x[m-P]和滤波器的系数b[0] ... b[P]求乘积和而得。而IIR滤波器只不过是再减去之前的输出y[m-1] ... y[m-Q]和系数a[1] ... a[m-Q]的乘积和。

总之，数字滤波器的计算方法并不复杂，仅仅是数组对应元素的乘积和求和而已。然而其计算量对于Python来说是相当大的：通常FIR滤波器的系数长度都上百，而CD音质的数字声音信号一秒钟有44100个取样值，假设滤波器的长度是100，那么一秒钟需要计算4百万次以上的乘积和加法。这对于Python这样的动态语言来说是很困难的。

因此scipy的signal库中提供了lfilter函数完成数字滤波器的计算工作。由于它是在C语言级别实现的，因此处理速度相当快：

```
signal.lfilter(b, a, x, axis=-1, zi=None)
```

其中的b和a是滤波器的系数，x是输入。lfilter函数并不是直接使用上面的IIR滤波器计算公式进行计算，而是对其进行了如下的变形：

```
y[m] = b[0]*x[m] + z[0,m-1]                           (1)
z[0,m] = b[1]*x[m] + z[1,m-1] - a[1]*y[m]             (2)
...
z[n-3,m] = b[n-2]*x[m] + z[n-2,m-1] - a[n-2]*y[m]
z[n-2,m] = b[n-1]*x[m] - a[n-1]*y[m]
```

这段公式就没有那么直白了，但是只需要仔细的观察一下就不难发现，将式(2)的时间变为m-1，得到：

```
z[0,m-1] = b[1]*x[m-1] + z[1,m-2] - a[1]*y[m-1]     (3)
```

将其带入到式(1)中，发现b[0]*x[m], b[1]*x[m-1], - a[1]*y[m-1]等项已经和IIR公式中的一致，依次如此代入下去最后得到的公式和IIR滤波器的公式是一致的。这个计算公式被称为直接2型。

直接1型的公式中，为了计算m时刻的输出y[m]，除了需要m时刻的输入x[m]之外，还需要x[m-1]到x[m-P]和y[m-1]到y[m-Q]，这些值都需要被作为滤波器的内部状态保存起来，因此需要保存P+Q个数。而根据直接2型的公式，只需要保存n-1个数z[0]到z[n-2]，其中n为 max(len(a), len(b))，即max(P, Q)。数组z就是滤波器的状态。

滤波器的初始状态通过关键字参数zi传到lfilter函数，当zi不是None时，lfilter将返回滤波器的最终状态zf，于是其返回值为(y, zf)，如果zi为None的话，那么只返回滤波器的输出y。

当使用lfilter对很长的输入进行滤波计算时，不能一次把数据都读入到数组x中，因此需要对数据进行分段滤波，这时就需要将上一次调用lfilter时返回的数组zf，传递到下一次lfilter函数调用。下面的程序演示了这种分段滤波的方法：

```
# -*- coding: utf-8 -*-
import scipy.signal as signal
import numpy as np
import pylab as pl

# 某个均衡滤波器的参数
a = np.array([1.0, -1.947463016918843, 0.9555873701383931])
b = np.array([0.9833716591860479, -1.947463016918843, 0.97221571095

# 44.1kHz, 1秒的频率扫描波
t = np.arange(0, 0.5, 1/44100.0)
x= signal.chirp(t, f0=10, t1 = 0.5, f1=1000.0)

# 直接一次计算滤波器的输出
y = signal.lfilter(b, a, x)

# 将输入信号分为50个数据一组
x2 = x.reshape((-1,50))

# 滤波器的初始状态为0, 长度是滤波器系数长度-1
z = np.zeros(max(len(a),len(b))-1, dtype=np.float)
y2 = [] # 保存输出的列表

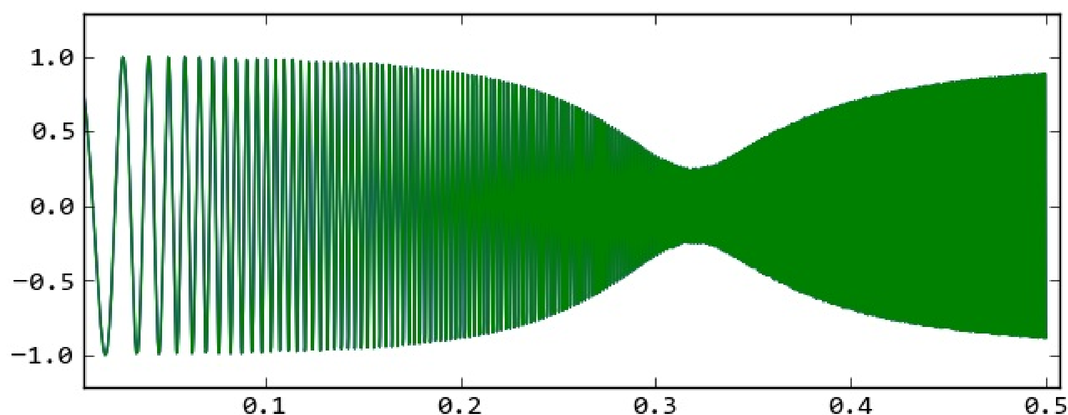
for tx in x2:
    # 对每段信号进行滤波，并更新滤波器的状态z
    ty, z = signal.lfilter(b, a, tx, zi=z)
    # 将输出添加到输出列表中
    y2.append(ty)

# 将输出y2转换为一维数组
y2 = np.array(y2)
y2 = y2.reshape((-1,))

# 输出y和y2之间的误差
print np.sum((y-y2)**2)

# 绘图
pl.plot(t, y, t, y2)
pl.show()
```

程序所输出的误差为0，经过滤波器之后的频率扫描波形如下图所示：

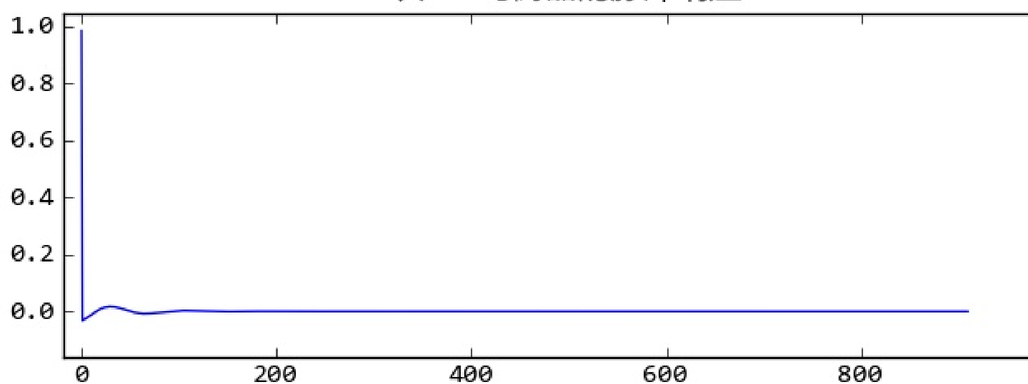


经过均衡滤波器之后的频率扫描波形

此程序中使用的IIR滤波器的系数为二次均衡滤波器的系数，其系数的设计算法将在下节进行介绍。为了观察滤波器的频率特性，我们让它对频率扫描波进行处理。分别采用一次滤波和分段滤波两种方式调用lfilter函数，我们看到两个结果完全一样。使用分段滤波结合pyaudio库，我们很容易写出对声卡采集的连续的声音信号进行滤波并输出的实时滤波程序。

如果将一个脉冲信号输入到滤波器中，所得到的输出被称为滤波器的其脉冲响应。所谓脉冲信号就是在时刻0为1，其余时刻均为0的信号。根据FIR滤波器的公式，FIR滤波器的脉冲响应就是滤波器的系数。而IIR滤波器的脉冲响应就不是很直观了，下面使用lfilter计算IIR滤波器的脉冲响应，其中的IIR滤波器的系数和前面的一样（在IPython或者Spyder中运行上面的程序之后，再输入下面的程序）：

```
>>> impulse = np.zeros(1000, dtype=np.float)
>>> impulse[0] = 1
>>> h = signal.lfilter(b, a, impulse)
>>> h[-1]
-4.2666825205952273e-12
```

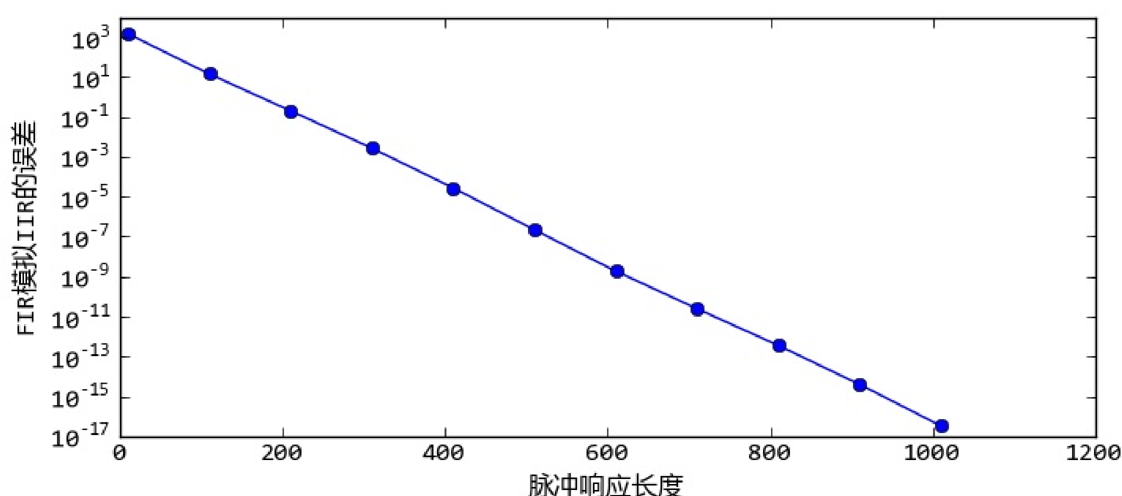


均衡滤波器的脉冲响应

如果你观察一下h的具体的值就会发现随着时间的推移，h越来越小，但是始终不会为0，其脉冲响应是无限长度的，因此才被称作无限脉冲响应滤波器。如果我们将h当作FIR滤波器的系数对信号x进行滤波的话，得到的结果

```
>>> y3 = signal.lfilter(h, 1, x)
>>> np.sum((y-y3)**2)
3.7835244127856444e-17
```

显然由于h是逐渐衰减的，只要我们测量足够长的脉冲响应，就可以用FIR滤波器足够精确地模拟IIR滤波器。下图显示的是误差和FIR滤波器的长度之间的关系。显然由于IIR滤波器的脉冲响应是呈指数衰减的，因此精度随着长度呈指数增加，请注意Y轴是对数坐标。



随着FIR滤波器的长度的增加误差呈指数减小

此图的计算程序如下，注意用lfilter计算FIR滤波器时，设置参数a的值为1：

```
# -*- coding: utf-8 -*-
import scipy.signal as signal
import numpy as np
import pylab as pl

# 某个均衡滤波器的参数
a = np.array([1.0, -1.947463016918843, 0.9555873701383931])
b = np.array([0.9833716591860479, -1.947463016918843, 0.97221571095

# 44.1kHz, 1秒的频率扫描波
t = np.arange(0, 0.5, 1/44100.0)
x= signal.chirp(t, f0=10, t1 = 0.5, f1=1000.0)
y = signal.lfilter(b, a, x)
ns = range(10, 1100, 100)
err = []

for n in ns:
    # 计算脉冲响应
    impulse = np.zeros(n, dtype=np.float)
    impulse[0] = 1
    h = signal.lfilter(b, a, impulse)

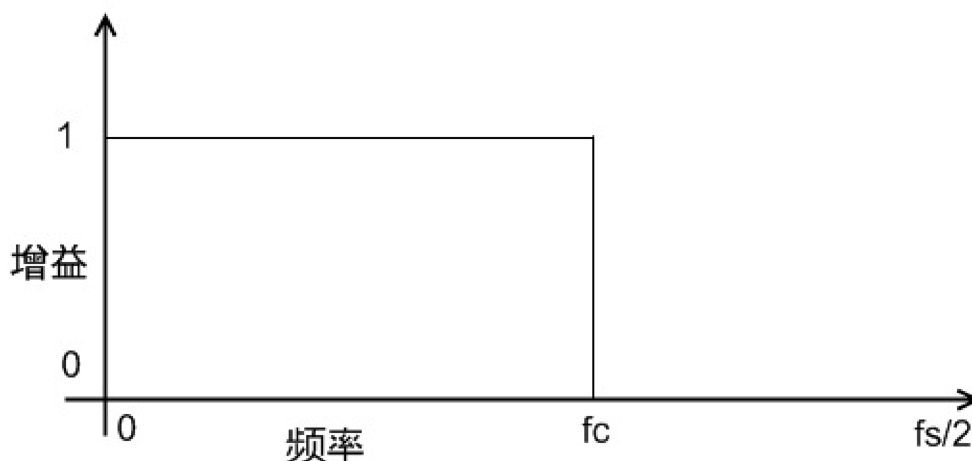
    # 直接FIR滤波器的输出
    y2 = signal.lfilter(h, 1, x)

    # 输出y和y2之间的误差
    err.append(np.sum((y-y2)**2))

# 绘图
pl.figure(figsize=(8,4))
pl.semilogy(ns , err, "-o")
pl.xlabel(u"脉冲响应长度")
pl.ylabel(u"FIR模拟IIR的误差")
pl.show()
```

FIR滤波器设计

理想的低通滤波器频率响应如下图所示：



理想低通滤波器的频率响应

其中 f_c 为阻带频率。通常为了计算方便，将取样频率正规化为1。于是 f_c 的含义就是每个取样点所包含的信号的周期数，例如0.1表示每个取样点包含0.1个周期，即一个周期有10个取样点。根据离散傅立叶变换的公式可以求出此理想低通滤波器的脉冲响应为：

$$h_{ideal}(n) = \frac{\sin(2\pi f_c)}{\pi n} = 2f_c sinc(2f_c n)$$

其中n为负无穷到正无穷的整数。显然此脉冲响应不但无限长，而且不满足因果律，因为输入信号在0时刻出现的脉冲，而输出信号却在0时刻之前就有值了。

这样的脉冲响应当然无法用FIR滤波器实现，一个最直观的近似方法就是取 h_{ideal} 中 0<=n<L 的L个值当作低通FIR滤波器的系数。下面的程序计算此低通滤波器的频率响应：

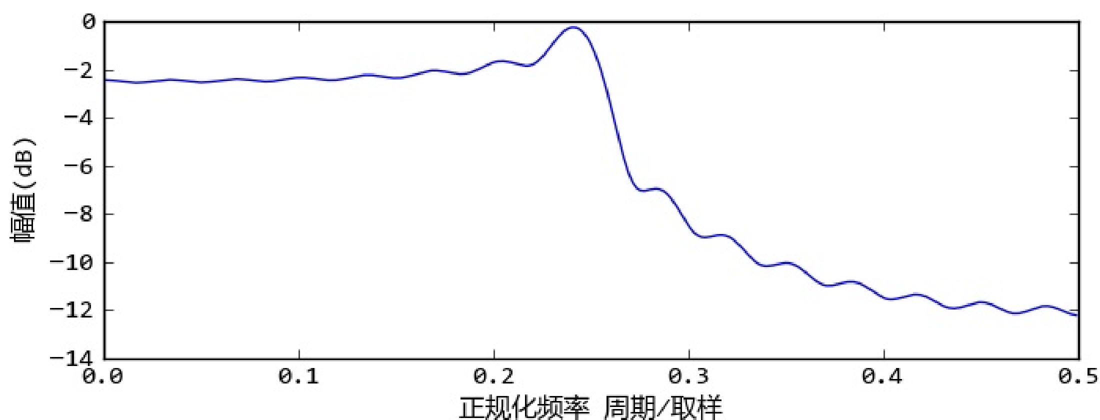
```
# -*- coding: utf-8 -*-
import scipy.signal as signal
import numpy as np
import pylab as pl

def h_ideal(n, fc):
    return 2*fc*np.sinc(2*fc*np.arange(-n, n, 1.0))

b = h_ideal(30, 0.25)

w, h = signal.freqz(b)

pl.figure(figsize=(8,3))
pl.plot(w/2/np.pi, 20*np.log10(np.abs(h)))
pl.xlabel(u"正规化频率 周期/取样")
pl.ylabel(u"幅值(dB)")
pl.show()
```



截取sinc函数正时间部分作为脉冲响应的低通滤波器

用freqz计算频率响应

freqz用于计算数字滤波器的频率响应，它的调用方式如下：

```
freqz(b, a=1, worN=None, whole=0, plot=None)
```

其中b和a是滤波器的系数，worN为所计算的频率点数，whole为0表示计算频率的上限为pi，whole为1表示计算频率的上限为2*pi。

它返回一个组元 (w,h)，其中w为所有计算了响应的频率数组，其值为正规化的圆频率，因此通过w/(2*pi)可以计算出对应的正规化频率。h是一个复数数组，它表示滤波器系统在每个对应的频率点的响应。复数的幅值表示滤波器的增益特性，相角表示滤波器的相位特性。

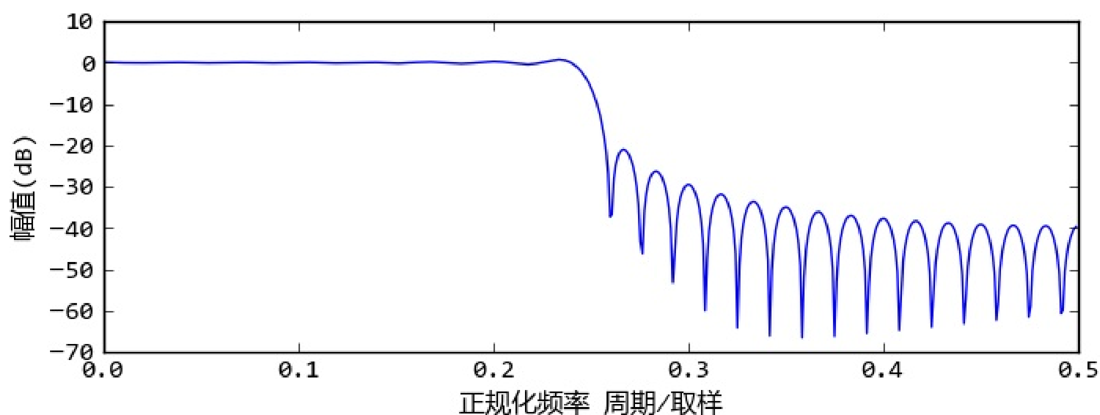
程序中使用freqz计算滤波器的频率响应，并用 20*np.log10(np.abs(h)) 计算h以dB衡量的幅值。

用firwin设计滤波器

显然此频率响应和理想的低通滤波器相差甚远，并且即使增加FIR滤波器的系数也没有作用。因为我们舍弃了n<0的那一半系数，而这些系数有着相当大的影响，因此只截取n>=0的部分是不够的，如果我们将n<0的那一半系数也添加进滤波器的话，得到的频率响应将会有很大的改善。如下重新定义h_ideal函数，它返回h_{ideal}中-n到n之间的系数：

```
def h_ideal(n, fc):
    return 2*fc*np.sinc(2*fc*np.arange(-n, n, 1.0))
```

下面是添加n<0系数之后的频率响应：



对称截取sinc函数的低通滤波器

这样做虽然改善了频率响应，但是给系统带来了许多延时，为了频率响应更好必须增加滤波器的点数，然而为了减少延时，必须减少点数，为了解决这个矛盾，我们给系数乘上一个窗函数，让它快速收敛。

SciPy提供了firwin用窗函数设计低通滤波器，firwin的调用形式如下：

```
firwin(N, cutoff, width=None, window='hamming')
```

其中N为滤波器的长度；cutoff为以$f_s/2$正规化的频率；window为所使用的窗函数。

下面的程序用firwin设计低通滤波器，并且和上面的结果进行比较，注意由于firwin的cutoff频率是以取样频率/2正规化的，因此它是前面所介绍的f_c的两倍。

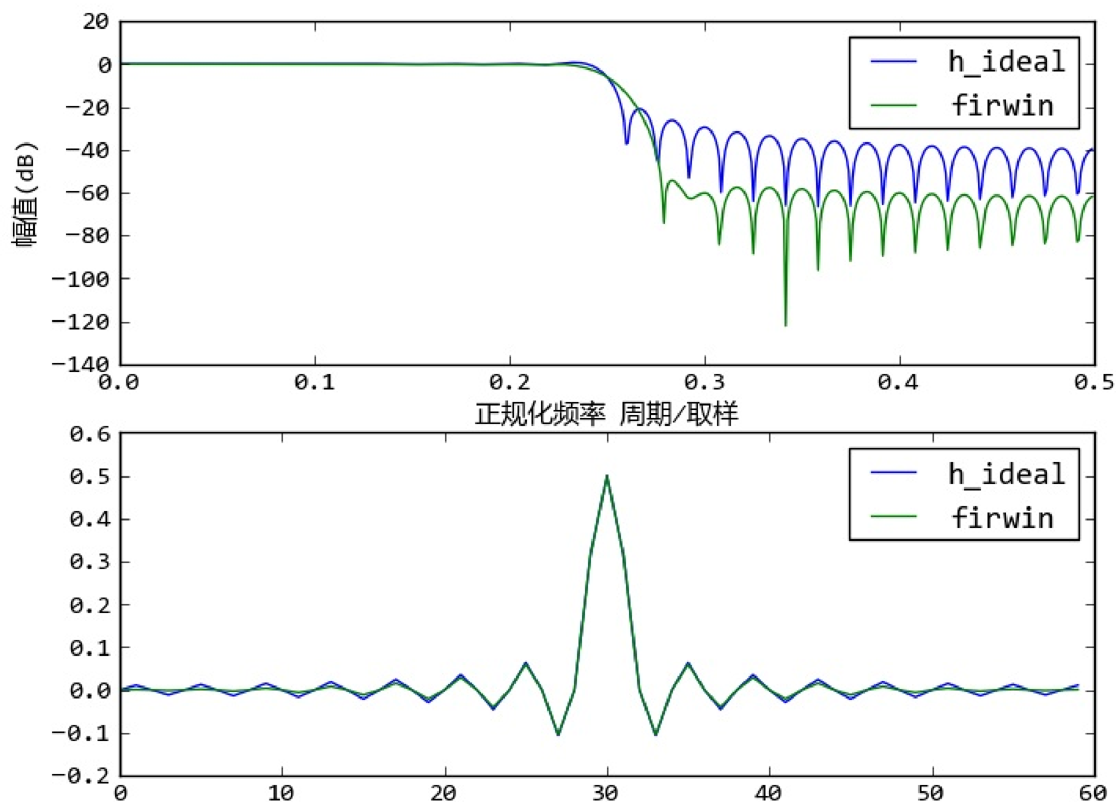
```
# -*- coding: utf-8 -*-
import scipy.signal as signal
import numpy as np
import pylab as pl

def h_ideal(n, fc):
    return 2*fc*np.sinc(2*fc*np.arange(-n, n, 1.0))

b = h_ideal(30, 0.25) # 以fs正规化的频率
b2 = signal.firwin(len(b), 0.5) # 以fs/2正规化的频率

w, h = signal.freqz(b)
w2, h2 = signal.freqz(b2)

pl.figure(figsize=(8,6))
pl.subplot(211)
pl.plot(w/2/np.pi, 20*np.log10(np.abs(h)), label=u"h_ideal")
pl.plot(w2/2/np.pi, 20*np.log10(np.abs(h2)), label=u"firwin")
pl.xlabel(u"正规化频率 周期/取样")
pl.ylabel(u"幅值(dB)")
pl.legend()
pl.subplot(212)
pl.plot(b, label=u"h_ideal")
pl.plot(b2, label=u"firwin")
pl.legend()
pl.show()
```



firwin使用窗函数设计的低通滤波器的频率响应和脉冲响应

使用firwin函数设计的滤波器并不是最优化的，为了实现同样效果频率响应，还存在长度更短的FIR滤波器。

用remez设计滤波器

remez函数能够帮助我们找到更优的滤波器系数。remez的调用形式如下：

```
remez(numtaps, bands, desired,
    weight=None, Hz=1, type='bandpass', maxiter=25, grid_density=1
```

其中：

- **numtaps**：所设计的FIR滤波器的长度
- **bands** ： 一个递增序列，它包括频率响应中的所有频带的边界，其值在0到Hz/2之间，如果参数Hz为缺省值1的话，那么可以把它当作是以取样频率正规化的频率
- **desired**：长度为bands的一半的增益序列，它给出频率响应在bands中的每个频带的增益值
- **weight**：长度和desired一样的权重序列，它给出desired中的每个增益所占的权重，即给出desired中的每个增益的重要性，值越大表示其越重要

- **type** : 'bandpass'或者'differentiator'，本书只介绍type为'bandpass'的情况

remez算法

remez是一种迭代算法，它能够找到一个n阶多项式，使得在指定的区间中此多项式和指定函数之间的最大误差最小化。由于FIR滤波器的频率响应实际上是一个多项式函数（请参考下节内容），因此可以用remez算法进行FIR滤波器系数设计。

remez返回经过remez算法最优化之后的FIR滤波器的系数。此系数和用firwin所设计的结果一样是对称的。当numtaps为偶数时，所设计的滤波器对于取样频率/2的响应为0，因此无法设计出长度为偶数的高通滤波器。

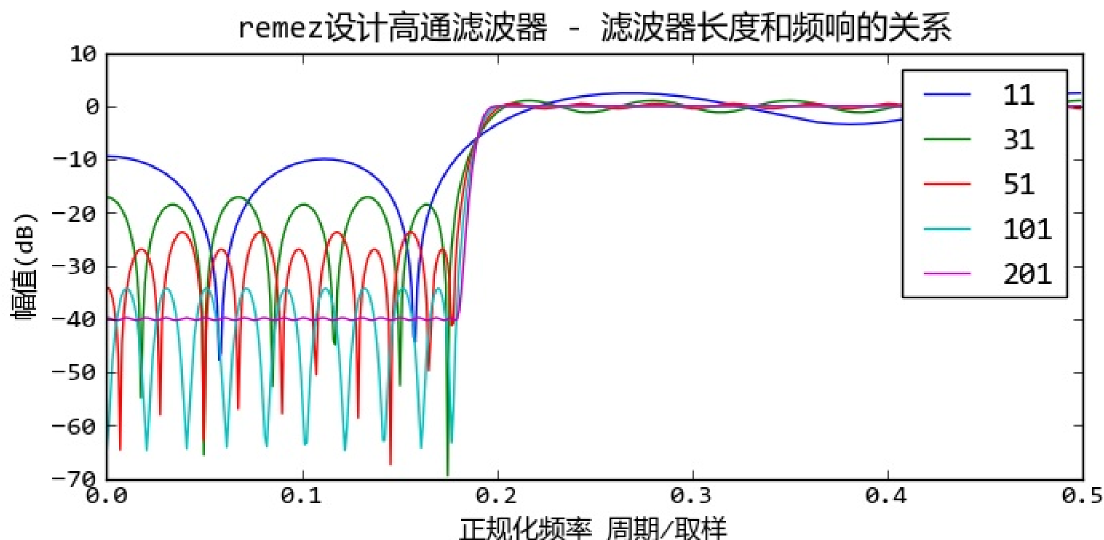
下面的程序演示通过remez设计高通滤波器：

```
# -*- coding: utf-8 -*-
import scipy.signal as signal
import numpy as np
import pylab as pl

for length in [11, 31, 51, 101, 201]:
    b = signal.remez(length, (0, 0.18,  0.2,  0.50), (0.01, 1))
    w, h = signal.freqz(b, 1)
    pl.plot(w/2/np.pi, 20*np.log10(np.abs(h)), label=str(length))
pl.legend()
pl.xlabel(u"正规化频率 周期/取样")
pl.ylabel(u"幅值(dB)")
pl.title(u"remez设计高通滤波器 - 滤波器长度和频响的关系")
pl.show()
```

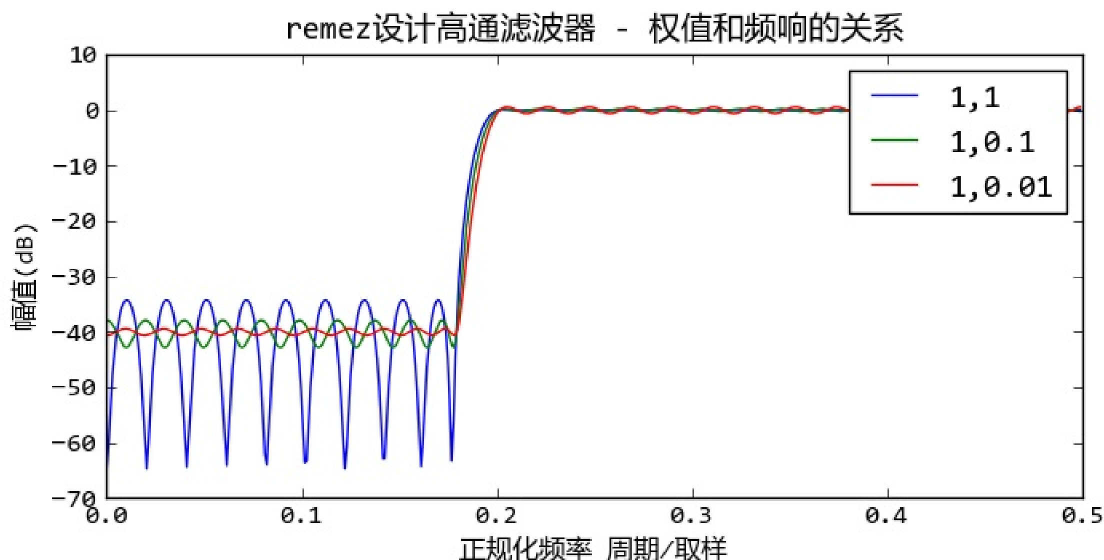
程序中，remez函数的bands参数给出两个频带(以取样频率正规化)：0到0.18和0.2到0.5，而desired给出两个频带的增益分别为0.01和1，因此它所设计的是一个通带频率为0.2、阻带增益为-40dB的高通滤波器。

此程序显示出滤波器长度和频率响应之间存在如下关系，可以看出滤波器越长，频率响应越接近设计值：



remez设计的高通滤波器，长度越长频率响应越接近设计值

下图显示权值和频率响应之间的关系，图中的滤波器长度为101。我们注意到，当权值为1, 0.01(红色曲线)时，两个频带的增益抖动量相同，这个权值正好和增益desired的设置相反。这时因为缺省情况下，增益越大的频带的频率响应要求越精确，而当权值和增益的乘积相等时，频率响应的误差也就相同了。



remez设计滤波器时权值影响频率响应

滤波器级联

假设有两个滤波器h1和h2，我们将h1的输出输入到h2，这样得到的滤波器称为h1和h2的级联。级联后的滤波器的脉冲响应为h1和h2的脉冲响应的卷积，而其频率响应为两个滤波器的频率响应的乘积。

下面的程序先用remez分别设计一个高通滤波器h1和一个低通滤波器h2，然后通过卷积计算出它们的级联滤波器h3的系数：

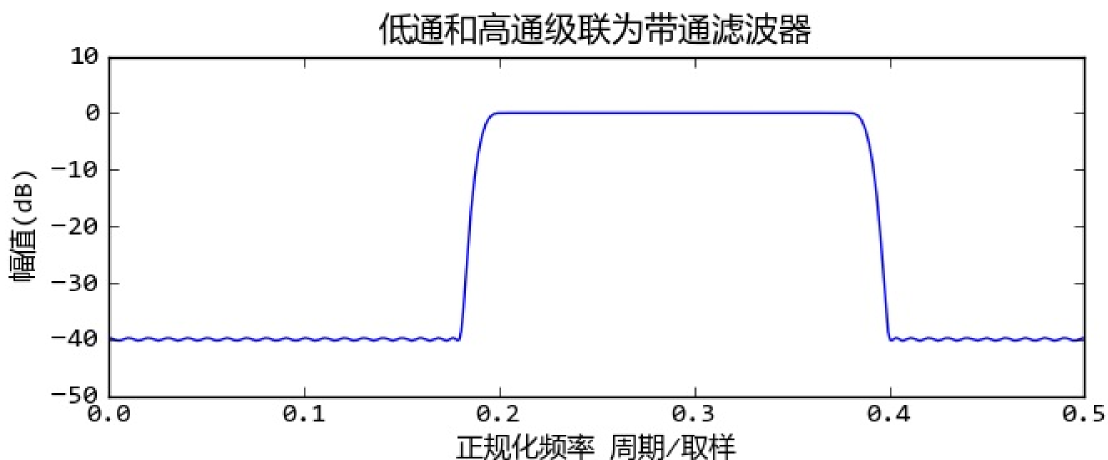
```
# -*- coding: utf-8 -*-
import scipy.signal as signal
import numpy as np
import pylab as pl

h1 = signal.remez(201, (0, 0.18,  0.2,  0.50), (0.01, 1))
h2 = signal.remez(201, (0, 0.38,  0.4,  0.50), (1, 0.01))
h3 = np.convolve(h1, h2)

w, h = signal.freqz(h3, 1)
pl.plot(w/2/np.pi, 20*np.log10(np.abs(h)))

pl.legend()
pl.xlabel(u"正规化频率 周期/取样")
pl.ylabel(u"幅值(dB)")
pl.title(u"低通和高通级联为带通滤波器")
pl.show()
```

最后使用freqz函数计算h3的频率响应：



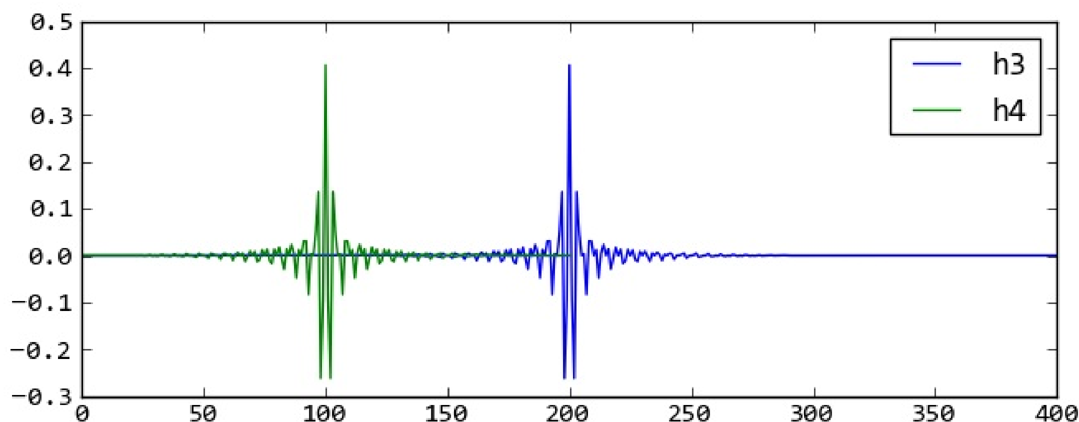
低通和高通滤波器级联之后是带通滤波器

可以看出，所得到的是一个带通滤波器。

我们也可以直接用remez设计带通滤波器：

```
>>> h4 = signal.remez(201, (0, 0.18, 0.2, 0.38, 0.4, 0.50), (0.01,
```

如果你观察此滤波器的频率响应的话，发现它和h3的基本一致，如果比较h3和h4的话，我们得到如下结果：



级联的滤波器和remz设计的带通滤波器的脉冲响应近似

可以看出虽然h3的长度几乎是h4的两倍，但是由于它的许多系数都接近于0，因此h3和h4的频率响应近似相同。

IIR滤波器设计

通常在设计数字IIR滤波器时，都会先设计一个对应的模拟滤波器，然后通过双线性变换将模拟滤波器转换为数字滤波器。这意味着我们需要在s复平面上设计滤波器的传递函数H(s)。当H(s)的所有的极点都在s的左半平面时，滤波器的响应是稳定的。下面以巴特沃斯滤波器为例，说明这一设计过程。

巴特沃斯低通滤波器

巴特沃斯低通滤波器的振幅的平方和频率之间的关系可以用如下公式表示：

$$|H(j\omega)|^2 = \frac{1}{1+\left(\frac{\omega}{\omega_c}\right)^{2n}}$$

其中n为滤波器的阶数，ω_c为振幅下降3dB时的截止频率。这个公式很容易理解：

- 当ω越小，振幅越接近于1
- 当ω越大，振幅越接近于0
- 随着n的增大，振幅接近于1或者0的速度将变快，即n越大，低通滤波器在阻频带的衰减速度将越快
- 当$\omega=\omega_c$时，振幅的平方为1/2，即-3dB

下面我们推导出巴特沃斯低通滤波器的传递函数H(s)，其中$s=\delta+j\omega$，为复数平面上的点。

由于当$H(s)H(-s)=|H(j\omega)|^2$，因此将$\omega=s/j$带入到巴特沃斯低通滤波器的振幅平方公式中可以得到：

$$H(s)H(-s) = \frac{{G_0}^2}{1+\left(\frac{-s^2}{\omega_c^2}\right)^n}$$

此公式有2n个极点，其中n个在左半平面，n个在右半平面，由于H(s)必须是稳定的，因此左半平面的n的极点属于H(s)。

最后得到的传递函数为：

$$H(s) = \frac{1}{\prod_{k=1}^{n}(s - s_k)/\omega_c}$$

其中s_k为左半平面上的极点：

$$s_k = \omega_c e^{\frac{j(2k+n-1)\pi}{2n}} \qquad \mathrm{k} = 1, 2, 3, \ldots, \mathrm{n}$$

下面的程序绘制6、7阶巴特沃斯低通滤波器的S复平面上的极点：

```
# -*- coding: utf-8 -*-
from scipy import signal
import numpy as np
import matplotlib.pyplot as pl

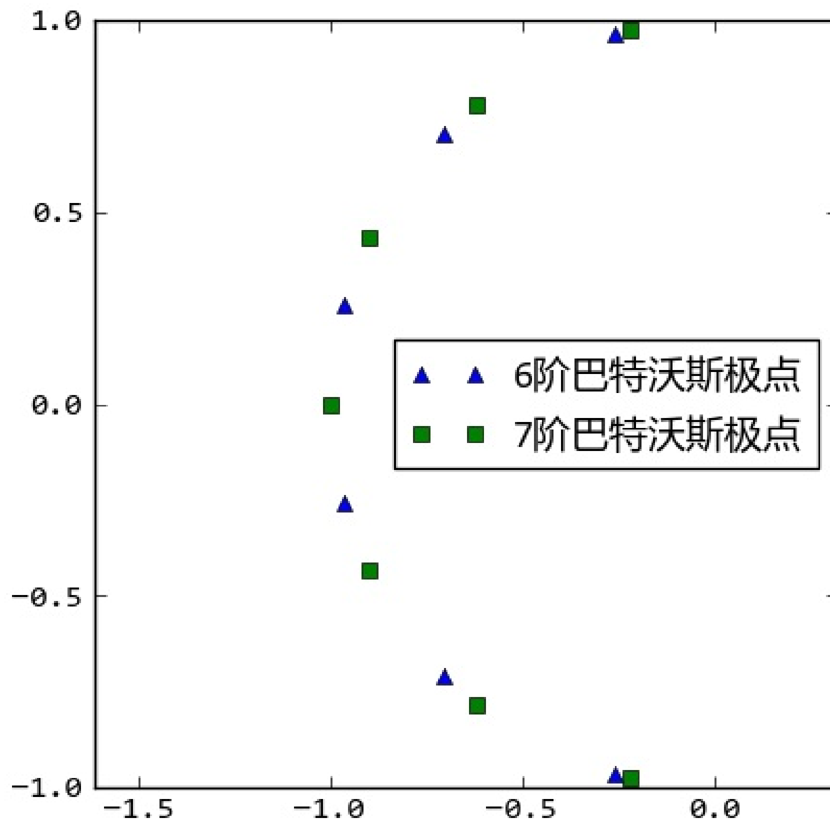
pl.figure(figsize=(5,5))

b, a = signal.butter(6, 1.0, analog=1)
z,p,k = signal.tf2zpk(b, a)
pl.plot(np.real(p), np.imag(p), '^', label=u"6阶巴特沃斯极点")

b, a = signal.butter(7, 1.0, analog=1)
z,p,k = signal.tf2zpk(b, a)
pl.plot(np.real(p), np.imag(p), 's', label=u"7阶巴特沃斯极点")

pl.axis("equal")
pl.legend(loc="center right")
pl.show()
```

程序中，使用butter函数设计巴特沃斯滤波器，缺省情况下它设计的是数字滤波器，为了设计模拟滤波器，需要传递关键字参数analog=1。获得传递函数的b和a的系数之后，通过tf2zpk函数将它们转换为零点和极点：



巴特沃斯低通滤波器在S复平面上的极点分布

双线性变换

有了连续时间的传递函数H(s)，下一步就是如何将它转换为离散时间的传递函数H(z)。转换的方法有几种，其中最常用的是双线性变换，其变换公式为：

$$s \leftarrow \frac{2}{T}\frac{z-1}{z+1}$$

其中T为离散时间的取样周期。双线性变换公式的推导过程请参考下面的链接：

双线性变换公式推导： http://en.wikipedia.org/wiki/Bilinear_transform

双线性变换实际上是s复平面和z复平面上的点的映射变换，他将s复平面上的竖线变换成z复平面上的圆，而s复平面上的Y轴对应于z复平面上的单位圆。下面的程序演示了这一对应关系：

```
# -*- coding: utf-8 -*-
import numpy as np
import pylab as pl

def stoz(s):
    """
 将s复平面映射到z复平面
 为了方便起见，假设取样周期T=1
 """
    return (2+s)/(2-s)

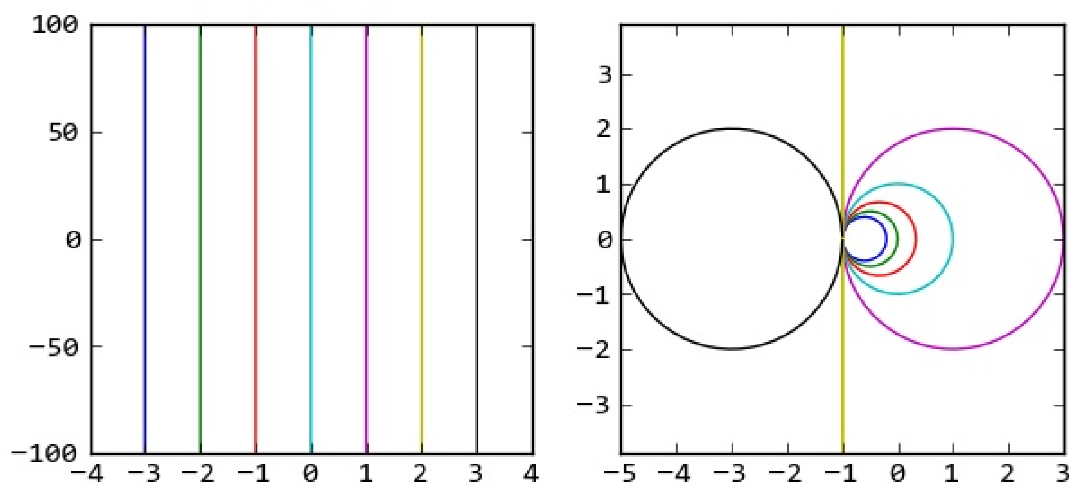
def make_vline(x):
    return x + 1j*np.linspace(-100.0,100.0,20000)

fig = pl.figure(figsize=(7,3))
axs = pl.subplot(121)
axz = pl.subplot(122)
for x in np.arange(-3, 4, 1):
    s = make_vline(x)
    z = stoz(s)
    axs.plot(np.real(s), np.imag(s))
    axz.plot(np.real(z), np.imag(z))

axs.set_xlim(-4,4)
axz.axis("equal")
axz.set_ylim(-3,3)

pl.show()
```

程序中的stoz函数是将s变换为z的变换公式，只需要对上述的双线性变换公式稍作变形即可得到，这里为了方便起见，假设取样周期T=1。



双线性变换将s平面(左图)上的竖线变换为z平面上的圆(右图)

通过双线性变换之后，滤波器的频率响应会发生变化。在下一节中我们会介绍，离散时间的滤波器的频率响应是将其传递函数H(z)用$z = e^{j\omega T}$进行替换。将其带入到双线性变换公式得到：

$$s = \frac{2}{T}\frac{z-1}{z+1} = \frac{2}{T}\frac{e^{j\omega T}-1}{e^{j\omega T}+1} = \frac{2}{T}\frac{e^{j\omega T/2}-e^{-j\omega T/2}}{e^{j\omega T/2}+e^{-j\omega T/2}} = j\frac{2}{T}\tan\left(\omega T/2\right)$$

对于s平面上的点来说，过原点的竖线ω通过如下的公式转换为连续时间的频率ω_a:

$$\omega_a = \frac{2}{T}\tan\left(\omega\frac{T}{2}\right)$$

而其反函数为：

$$\omega = \frac{2}{T}\arctan\left(\omega_a\frac{T}{2}\right)$$

下面让我们用程序来验证这个频率转换公式。首先我们载入所需要的库，并且定义离散时间的取样频率fs为8kHz，设计的巴特沃斯低通滤波器通带截至频率为1kHz：

```
>>> from scipy import signal
>>> from numpy import *
>>> fs = 8000.0
>>> f = 1000.0
```

下面使用butter函数设计一个3阶的巴特沃斯滤波器，注意关键字参数analog=1，表示设计连续时间传递函数H(s)的系数，由于通带频率参数为圆频率，因此需要乘以2*pi：

```
>>> b, a = signal.butter(3, 2*pi*f, analog=1)
```

模拟滤波器的系数b和a和H(s)的关系

假设b和a的长度分别为M和N，模拟滤波器的系数中b[0]为分子中s^{N-1}，b[-1]和a[-1]为分子分母的常数项的系数，即：

$$H(s) = \frac{b_0 s^{M-1} + b_1 s^{M-2} + \ldots + b_{M-1}}{a_0 s^{N-1} + a_1 s^{N-2} + \ldots + a_{N-1}}$$

然后调用双线性变换函数bilinear，将系数转换为离散时间的传递函数系数，通过关键字参数fs指定取样频率：

```
>>> b2, a2 = signal.bilinear(b,a,fs=fs)
```

接下来调用freqz函数得到此数字滤波器的频率响应，为了得到尽可能精确的值，我们通过worN关键字参数让它计算10000点的频率响应：

```
>>> w2, h2 = signal.freqz(b2,a2,worN=1000)
```

接下来将h2转换为增益，并且找到增益为-3dB(精确值为10*log10(0.5))*时所对应的正规化圆频率*w*的下标*idx*，*w/(*2pi)*fs就是其对应的实际频率值：

```
>>> p2 = 20*log10(abs(h2))
>>> idx = argmin(abs(p2-10*log10(0.5)))
>>> w2[idx]/2/pi*8000
952.8
```

通过频率转换公式得到的频率为：

```
>>> 2*fs*arctan(2*pi*f/2/fs) /2/pi
952.8840223
```

实际使用scipy.signal库设计IIR滤波器没有这么麻烦，因为它所提供的滤波器设计函数缺省都是直接设计数字滤波器。这些函数设计数字滤波器时采用的取样频率为2，即以香农频率fs/2为1进行正规化。因此要设计取样频率为fs、通带频率为f的滤波器需要将通带频率正规化为f/(fs/2)，下面调用butter函数设计数字低通滤波器，这里使用上述计算所得的通带频率：

```
>>>  b3,a3 = signal.butter(3, 952.8840223/(fs/2))
>>> sum(abs(b3-b2))
1.3226225670237568e-13
>>> sum(abs(a3-a2))
7.0876637892069994e-13
```

数字滤波器的系数b和a和H(z)的关系

假设b和a的长度分别为M和N，数字滤波器的系数中b[0]和a[0]分别为分子分母中常数项的系数，a[-1]为分母中$z^{-(M-1)}$的系数，即：

$$H(s) = \frac{b_0 + b_1 z^{-1} + \ldots + b_{M-1} z^{-(M-1)}}{a_0 + a_1 z^{-1} + \ldots + a_{N-1} z^{-(N-1)}}$$

所得的滤波器的系数b3和a3与手工通过bilinear函数计算的系数b2和a2是一致的。在signal库设计数字滤波器时，其内部会先通过频率转换公式对频率进行转换，然后设计连续时间的传递函数系数，最后通过bilinear函数进行系数转换。有兴趣的读者可以查看signal.iirfilter函数的源代码。

滤波器的频带转换

只要知道了低通滤波器的传递函数H(s)，就很容易利用变量替换设计出同样阶数的高通、带通或者其它通带频率的低通滤波器。让我们来看看低通滤波器的变换。

假设我们使用巴特沃斯低通滤波器设计公式，设计出通带频率为1弧度/秒的标准低通滤波器：

```
>>> b, a = signal.butter(2, 1.0, analog=1)
>>> np.real(b)
array([ 1.])
>>> np.real(a)
array([ 1\.        ,  1.41421356,  1\.        ])
```

$$H(s) = \frac{1}{s^2 + 1.4142s + 1}$$

为了让它变为通带频率为ω_c的低通滤波器，只需要进行如下替换：

$$s \to \frac{s}{\omega_c}$$

由于当$s = j$时振幅下降3dB的，而$s = j\omega_c$处下降3dB。下面是通带频率为2弧度/秒的2阶低通滤波器的系数：

```
>>> b2, a2 = signal.butter(2, 2.0, analog=1)
>>> np.real(b2)
array([ 4.])
>>> np.real(a2)
array([ 1\.        ,  2.82842712,  4\.        ])
```

可以看出将$s \to \frac{s}{2}$代入到前面的H(s)中即可得到这些系数。

低通滤波器转高通滤波器的替代公式为：

$$s \to \frac{\omega_c}{s}$$

此替代公式很容易理解：

- 若ω为0，则替代之后的频率为无穷大，而低通滤波器无穷大处的频率响应为0，即转换之后的滤波器在0处的频率响应为0；
- 若ω为无穷大，则替代之后的频率为0，因此转换之后的滤波器在无穷大处频率响应为1。

下面设计通带频率为1弧度/秒的高通滤波器：

```
>>> b3,a3 = signal.butter(2,1.0,btype="high",analog=1)
>>> np.real(b3)
array([ 1.,  0.,  0.])
>>> np.real(a3)
array([ 1\.        ,  1.41421356,  1\.        ])
```

可以看出这些系数是将s^2之后得到的。

低通滤波器还可以转换为带通滤波器，这可能有点难以理解，让我们先来看看替代公式，假设带通滤波器的高低通带频率为ω_1：

$$s \to \frac{\omega_0}{\Delta\omega}\left(\frac{s}{\omega_0}+\frac{\omega_0}{s}\right)$$

其中$\omega_0 = \sqrt{\omega_1\omega_2}$。$\omega_0$则是通带的中心频率。

让我们通过下面的程序来研究一下为何这种替代能够将低通映射为带通滤波器：

```
# -*- coding: utf-8 -*-
import numpy as np
from scipy import signal
import pylab as pl

b, a = signal.butter(2, 1.0, analog=1)

# 低通->带通的频率变换函数
w1 = 1.0 # 低通带频率
w2 = 2.0 # 高通带频率
dw = w2 - w1 # 通带宽度
w0 = np.sqrt(w1*w2) # 通带中心频率

# 产生10**-2到10**2的频率点
w = np.logspace(-2, 2, 1000)

# 使用频率变换公式计算出转换之后的频率
nw = np.imag(w0/dw*(1j*w/w0 + w0/(1j*w)))

_, h = signal.freqs(b, a, worN=nw)
h = 20*np.log10(np.abs(h))

pl.figure(figsize=(8,5))

pl.subplot(221)
pl.semilogx(w, nw) # X轴使用log坐标绘图
pl.xlabel(u"变换前圆频率(弧度/秒)")
pl.ylabel(u"变换后圆频率(弧度/秒)")

pl.subplot(222)
pl.plot(h, nw)
pl.xlabel(u"低通滤波器的频率响应(dB)")

pl.subplot(212)
pl.semilogx(w, h)
pl.xlabel(u"变换前圆频率(弧度/秒)")
pl.ylabel(u"带通滤波器的频率响应(dB)")

pl.subplots_adjust(wspace=0.3, hspace=0.3, top=0.95, bottom=0.14)

print "center:", w[np.argmin(np.abs(nw))]
pl.show()
```

程序中先使用butter函数设计一个模拟的二阶标准低通滤波器：

```
b, a = signal.butter(2, 1.0, analog=1)
```

我们要将其转换为通带频率为w1=1到w2=2的带通滤波器：

```
# 低通->带通的频率变换函数
w1 = 1.0 # 低通带频率
w2 = 2.0 # 高通带频率
dw = w2 - w1 # 通带带宽
w0 = np.sqrt(w1*w2) # 通带中心频率
```

假设我们关心的频率响应的频率段为0.01到100，使用logspace函数产生一个这个区间的等比数列w：

```
w = np.logspace(-2, 2, 1000)
```

通过带通频率转换公式将其转换为新的频率序列nw：

```
nw = np.imag(w0/dw*(1j*w/w0 + w0/(1j*w)))
```

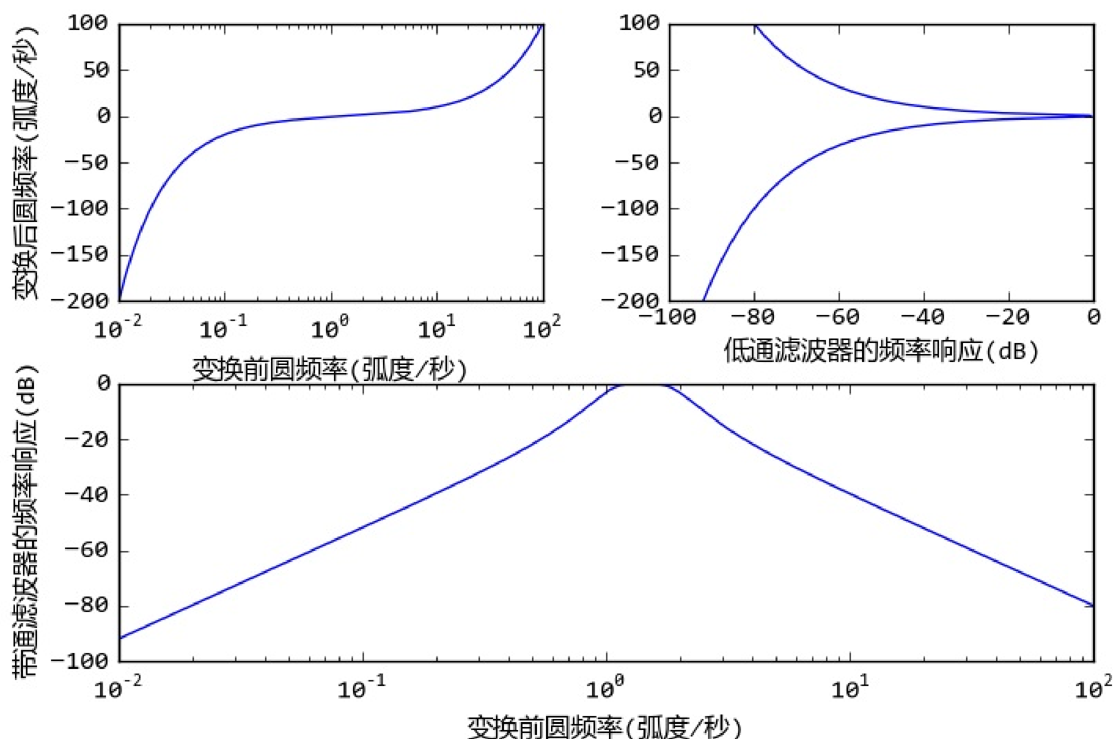
使用此新的频率序列nw计算出每个频率点对应的低通滤波器的频率响应，注意我们通过worN关键字传输让freqs函数计算指定频率的频率响应：

```
_, h = signal.freqs(b, a, worN=nw)
h = 20*np.log10(np.abs(h))
```

下面的语句就绘制出w和h的关系，也就是带通滤波器的频率响应：

```
pl.semilogx(w, h)
```

下图是程序的输出。其中左上图绘制的是频率转换函数，右上图绘制的是低通滤波器的频率响应(X轴为响应，Y轴为频率)，最下面绘制的是最终的带通滤波器的频率响应。



使用频率转换公式将低通变为带通滤波器

由于带通的频率转换公式将0到无穷大映射到正无穷到负无穷，而低通滤波器在正负无穷处的频率响应都为0，因此可以想象转换后的滤波器是一个带通滤波器。而转换之后的频率nw为0的点所对应的原始频率就是带通滤波器的中心频率，此处的频率响应为1，下面的程序找到nw绝对值最小的下标，并输出其对应的转换前的频率，我们看到它和w0是一致的：

```
>>> print w[np.argmin(np.abs(nw))]
1.4130259906
```

事实上，scipy.signal库已经为我们提供了频带转换的函数：

- lp2lp : 低通转低通
- lp2hp : 低通转高通
- lp2bp : 低通转带通
- lp2bs : 低通转带阻，转带阻的公式留给读者思考

下面以lp2bp为例简要说明一下函数的用法，假设b,a为二阶标准低通滤波器，下面的语句将转换为通带为1到2弧度的带通滤波器，前两个参数为滤波器的系数，后两个参数分别为中心频率和通带带宽：

```
>>> b, a = signal.butter(2, 1.0, analog=1)
>>> b3, a3 = signal.lp2bp(b,a,np.sqrt(2), 1)
```

我们也可以直接调用butter设计一个低通滤波器：

```
>>> b4, a4 = signal.butter(2, [1,2], btype='bandpass', analog=1)
```

两个结果是完全一致的：

```
>>> np.all(b3==b4)
True
>>> np.all(a3==a4)
True
```

滤波器的频率响应

前面的许多例子中都使用函数freqz计算滤波器的频率响应，在这一节中，让我们来深入研究一下freqz是如何计算频率响应的。在IPython中输入：

```
def freqz(b, a=1, worN=None, whole=0, plot=None):
    b, a = map(atleast_1d, (b,a))
    if whole:
        lastpoint = 2*pi
    else:
        lastpoint = pi
    if worN is None:
        N = 512
        w = numpy.arange(0,lastpoint,lastpoint/N)
    elif isinstance(worN, types.IntType):
        N = worN
        w = numpy.arange(0,lastpoint,lastpoint/N)
    else:
        w = worN
    w = atleast_1d(w)
    zm1 = exp(-1j*w)
    h = polyval(b[::-1], zm1) / polyval(a[::-1], zm1)
    if not plot is None:
        plot(w, h)
    return w, h
```

研究一下这段代码，不难发现真正的计算频率响应的代码可以用如下3行程序概括：

```
w = numpy.arange(0,pi,pi/N)
zm1 = exp(-1j*w)
h = polyval(b[::-1], zm1) / polyval(a[::-1], zm1)
```

为了弄清楚为什么这3行代码能够计算滤波器的频率响应，让我们先来学习一下相关的理论知识。

滤波器的频率响应由滤波器的传递函数给出，IIR滤波器的计算公式如下：

$$y[m] = b[0]x[m] + b[1]x[m-1] + \cdots + b[P]x[m-P]$$
$$- a[1]y[m-1] - a[2]y[m-2] - \cdots - a[Q]y[m-Q]$$

根据Z变换的相关公式，容易求得其传递函数为：

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b[0] + z^{-1}b[1] + z^{-2}b[2] + \cdots + z^{-M}b[P]}{1 + z^{-1}a[1] + z^{-2}a[2] + \cdots + z^{-N}a[Q]}$$

其中z为复数平面上的任意一点。当z为单位圆上的点，即 $H(\omega)$ 就是滤波器的频率响应。 $e^{j\omega}$ 正好绕复数平面单位圆转一圈。由于复数平面上下两个半平面的复数存在共轭关系，因此通常只需要求上半圆的频率响应，因此下面的语句将上半圆等分为N份：

```
w = numpy.arange(0,pi,pi/N)
```

然后计算w中每点对应的复数值 z^{-1}，注意这里将负号带入，于是传递函数的分子分母部分就都变成了zm1的多项式函数：

```
zm1 = exp(-1j*w)
```

最后带入传递函数的公式中计算出频率响应h：

```
h = polyval(b[::-1], zm1) / polyval(a[::-1], zm1)
```

polyval(p, x)函数对于数组x中的每个元素计算多项式p的值，其计算公式如下：

```
p[0]*(x**N-1) + p[1]*(x**N-2) + ... + p[N-2]*x + p[N-1]
```

由于滤波器系数b和a的顺序正好和polyval的多项式系数p的顺序相反，因此通过数组切片运算b[::-1]将滤波器的系数反转。由于数组zm1中的值都为复数，因此所得到的频率响应h的值也都是复数。复数的幅值对应于频率响应中的增益特性，而其相角对应于频率响应中相位特性。

freqz中在Z平面单位圆上所取的点是等距线性的，然而我们经常需要在绘制频率响应图表时要求频率坐标为对数坐标，对于对数坐标，等距的频率点会造成低频过疏，高频过密的问题，因此我们可以如下改造freqz函数，使其更适合计算对数频率坐标的频率响应。

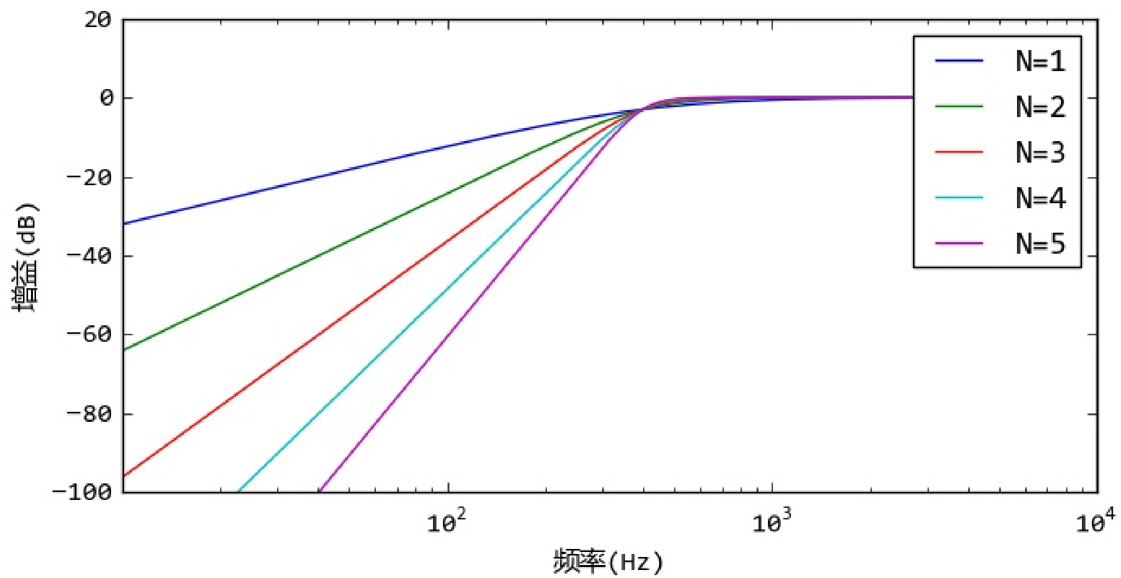
```
# -*- coding: utf-8 -*-
import numpy as np
import pylab as pl
import scipy.signal as signal

def logfreqz(b, a, f0, f1, fs, N):
    """
 以对数频率坐标计算滤波器b,a的频率响应
 f0, f1: 计算频率响应的开始频率和结束频率
 fs: 取样频率
 """
    w0, w1 = np.log10(f0/fs*2*np.pi), np.log10(f1/fs*2*np.pi)
    # 不包括结束频率
    w = np.logspace(w0, w1, N, endpoint=False)
    zm1 = np.exp(-1j*w)
    h = np.polyval(b[::-1], zm1) / np.polyval(a[::-1], zm1)
    return w/2/np.pi*fs, h

for n in range(1, 6):
    # 设计n阶的通频为0.1*4000 = 400Hz的高通滤波器
    b, a = signal.iirfilter(n, [0.1, 1])
    f, h = logfreqz(b, a, 10.0, 4000.0, 8000.0, 400)
    gain = 20*np.log10(np.abs(h))
    pl.semilogx(f, gain, label="N=%s" % n)
    slope = (gain[100]-gain[10]) / (np.log2(f[100]) - np.log2(f[10]
    print "N=%s, slope=%s dB" % (n, slope)
pl.ylim(-100, 20)
pl.xlabel(u"频率(Hz)")
pl.ylabel(u"增益(dB)")
pl.legend()
pl.show()
```

程序中的logfreqz函数计算系数为b和a的滤波器在f0到f1之间的频率响应，fs为取样频率，N为计算的点数。首先通过f/fs2pi将实际频率转换为与之对应的圆频率。然后通过logspace函数计算频率点的等比数列。最后和freqz一样通过调用polyval计算频率响应，返回值为实际频率点和对应的频率响应。

接下来通过调用iirfilter设计5个不同阶的IIR高通滤波器，通频为0.1，如果取样频率为8kHz的话，那么实际的通频为0.1*4kHz=400Hz。5个IIR滤波器的增益特性如下图所示：



iirfilter设计5个不同阶的IIR高通滤波器

由此图可知，随着IIR滤波器的阶数的增加，增益的下降速度增加，程序中第25行计算出下降处两个倍频之间的增益差值，其结果如下：

```
# -*- coding: utf-8 -*-

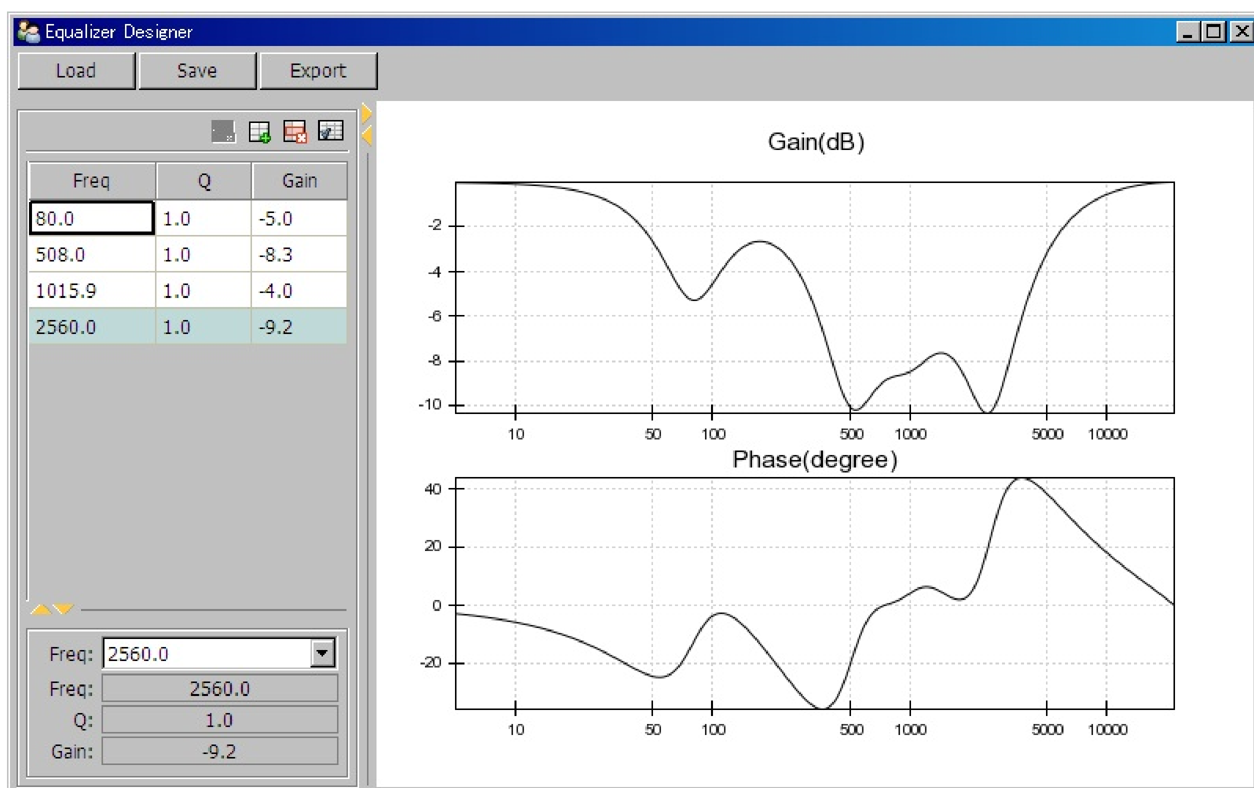
import scipy.signal as signal
import pylab as pl
import math
import numpy as np

def design_equalizer(freq, Q, gain, Fs):
    '''设计二次均衡滤波器的系数'''
    A = 10**(gain/40.0)
    w0 = 2*math.pi*freq/Fs
    alpha = math.sin(w0) / 2 / Q

    b0 = 1 + alpha * A
    b1 = -2*math.cos(w0)
    b2 = 1 - alpha * A
    a0 = 1 + alpha / A
    a1 = -2*math.cos(w0)
    a2 = 1 - alpha / A
    return [b0/a0,b1/a0,b2/a0], [1.0, a1/a0, a2/a0]

pl.figure(figsize=(8,4))
for freq in [1000, 2000, 4000]:
    for q in [0.5, 1.0]:
        for p in [5, -5, -10]:
            b,a = design_equalizer(freq, q, p, 44100)
            w, h = signal.freqz(b, a)
            pl.semilogx(w/np.pi*44100, 20*np.log10(np.abs(h)))
pl.xlim(100, 44100)
pl.xlabel(u"频率(Hz)")
pl.ylabel(u"振幅(dB)")
pl.subplots_adjust(bottom=0.15)
pl.show()
```

使用上节介绍的对数频率响应的求法以及TraitsUI和Chaco等界面库，我们可以设计如下界面的二次均衡器设计程序：



二次均衡器设计工具的界面

用户可以使用此程序添加、删除和编辑二次均衡器，并且即时查看均衡器级联之后的频率响应。完整的程序请参照： 二次均衡器设计

ScrubberEditor的BUG

如果界面中的ScrubberEditor无法用鼠标拖动修改的话，那么你需要修改site-packages目录下的 scrubber_editor.py 文件：

```
# Establish the slider increment:
increment = self.factory.increment
if increment <= 0.0:
    if (low is None) or (high is None) or isinstance( low, int ):
        increment = 1.0
    else:
        increment = pow( 10, round( log10( (high - low) / 100.0 )

self.increment = increment # ** 将此行移出if作用域
```

FFT演示程序

本章详细介绍如何综合利用之前所学习的numpy，traits，traitsUI和Chaco等多个库，编写一个FFT演示程序。此程序可以帮助你理解FFT是如何将时域信号转换为频域信号的，在开始正式的程序编写之前，让我们来复习一下有关FFT变换的知识，如果你没有学习过FFT，现在就是一个不错的学习机会。

FFT知识复习

FFT变换是针对一组数值进行运算的，这组数的长度N必须是2的整数次幂，例如64, 128, 256等等；数值可以是实数也可以是复数，通常我们的时域信号都是实数，因此下面都以实数为例。我们可以把这一组实数想像成对某个连续信号按照一定取样周期进行取样而得来，如果对这组N个实数值进行FFT变换，将得到一个有N个复数的数组，我们称此复数数组为频域信号，此复数数组符合如下规律：

- 下标为0和N/2的两个复数的虚数部分为0,
- 下标为i和N-i的两个复数共轭，也就是其虚数部分数值相同、符号相反。

下面的例子演示了这一个规律，先以rand随机产生有8个元素的实数数组x，然后用fft对其运算之后，观察其结果为8个复数，并且满足上面两个条件：

```
>>> x = np.random.rand(8)
>>> x
array([ 0.15562099,  0.56862756,  0.54371949,  0.06354358,  0.60678
 0.78360968,  0.90116887,  0.1588846 ])
>>> xf = np.fft.fft(x)
>>> xf
array([ 3.78195634+0.j        , -0.53575962+0.57688097j,
 -0.68248579-1.12980906j, -0.36656155-0.13801778j,
 0.63262552+0.j        , -0.36656155+0.13801778j,
 -0.68248579+1.12980906j, -0.53575962-0.57688097j])
```

FFT变换的结果可以通过IFFT变换（逆FFT变换）还原为原来的值：

```
>>> np.fft.ifft(xf)
array([ 0.15562099 +0.00000000e+00j,  0.56862756 +1.91940002e-16j,
 0.54371949 +1.24900090e-16j,  0.06354358 -2.33573365e-16j,
 0.60678158 +0.00000000e+00j,  0.78360968 +2.75206729e-16j,
 0.90116887 -1.24900090e-16j,  0.15888460 -2.33573365e-16j])
```

注意ifft的运算结果实际上是和x相同的，由于浮点数的运算误差，出现了一些非常小的虚数部分。

FFT变换和IFFT变换并没有增加或者减少信号的数量，如果你仔细数一下的话，x中有8个实数数值，而xf中其实也只有8个有效的值。

计算FFT结果中的有用的数值

由于虚数部共轭和虚数部为0等规律，真正有用的信息保存在下标从0到N/2的N/2+1个虚数中， 又由于下标为0和N/2的值虚数部分为0，因此只有N个有效的实数值。

下面让我们来看看FFT变换之后的那些复数都代表什么意思。

- 首先下标为0的实数表示了时域信号中的直流成分的多少
- 下标为i的复数a+b*j表示时域信号中周期为N/i个取样值的正弦波和余弦波的成分的多少， 其中a表示cos波形的成分，b表示sin波形的成分

让我们通过几个例子来说明一下，下面是对一个直流信号进行FFT变换：

```
>>> x = np.ones(8)
>>> x
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> np.fft.fft(x)/len(x)
array([ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.
 0.+0.j])
```

所谓直流信号，就是其值不随时间变化，因此我们创建一个值全为1的数组x，我们看到它的FFT结果除了下标为0的数值不为0以外，其余的都为0。(为了计算各个成分的能量多少，需要将FFT的结果除以FFT的长度)，这表示我们的时域信号是直流的，并且其成分为1。

下面我们产生一个周期为8个取样的正弦波，然后观察其FFT的结果：

```
>>> x = np.arange(0, 2*np.pi, 2*np.pi/8)
>>> y = np.sin(x)
>>> np.fft.fft(y)/len(y)
array([  1.42979161e-18 +0.00000000e+00j,
 -4.44089210e-16 -5.00000000e-01j,
 1.53075794e-17 -1.38777878e-17j,
 3.87737802e-17 -1.11022302e-16j,
 2.91853672e-17 +0.00000000e+00j,
 0.00000000e+00 -9.71445147e-17j,
 1.53075794e-17 +1.38777878e-17j,   3.44085112e-16 +5.00000000e-01j
```

如何用linspace创建取样点x

要计算周期为8个取样的正弦波，就需要把0-2*pi*的区间等分为*8*分，如果用*np.linspace(0, 2*np.pi, 8)的话，产生的值为：

```
>>> np.linspace(0, 2*np.pi, 8)
array([ 0\.         ,  0.8975979 ,  1.7951958 ,  2.6927937 ,  3.5903
 4.48798951,  5.38558741,  6.28318531])
>>> 2*np.pi / 0.8975979
7.0000000079986666
```

可以看出上面用linspace只等分为7份，如果要正确使用np.linspace的话， 可以如下调用，产生9个点，并且设置endpoint=False，最终结果不包括最后的那个点：

```
>>> np.linspace(0, 2*np.pi, 9, endpoint=False)
array([ 0\.         ,  0.6981317 ,  1.3962634 ,  2.0943951 ,  2.7925
 3.4906585 ,  4.1887902 ,  4.88692191,  5.58505361])
```

让我们再来看看对正弦波的FFT的计算结果吧。可以看到下标为1的复数的虚数部分为-0.5，而我们产生的正弦波的放大系数(振幅)为1，它们之间的关系是-0.5*（-2）=1。再来看一下余弦波形:

```
>>> np.fft.fft(np.cos(x))/len(x)
array([ -4.30631550e-17 +0.00000000e+00j,
 5.00000000e-01 -2.52659764e-16j,
 1.53075794e-17 +0.00000000e+00j,
 1.11022302e-16 +1.97148613e-16j,
 1.24479962e-17 +0.00000000e+00j,
 -1.11022302e-16 +1.91429446e-16j,
 1.53075794e-17 +0.00000000e+00j,   5.00000000e-01 -1.35918295e-16j
```

只有下标为1的复数的实数部分有有效数值0.5，和余弦波的放大系数1之间的关系是0.5*2=1。再来看2个例子：

```
>>> np.fft.fft(2*np.sin(2*x))/len(x)
array([  6.12303177e-17 +0.00000000e+00j,
 6.12303177e-17 +6.12303177e-17j,
 -1.83690953e-16 -1.00000000e+00j,
 6.12303177e-17 -6.12303177e-17j,
 6.12303177e-17 +0.00000000e+00j,
 6.12303177e-17 +6.12303177e-17j,
 -1.83690953e-16 +1.00000000e+00j,   6.12303177e-17 -6.12303177e-17
>>> np.fft.fft(0.8*np.cos(2*x))/len(x)
array([ -2.44921271e-17 +0.00000000e+00j,
 -3.46370983e-17 +2.46519033e-32j,
 4.00000000e-01 -9.79685083e-17j,
 3.46370983e-17 -3.08148791e-32j,
 2.44921271e-17 +0.00000000e+00j,
 3.46370983e-17 -2.46519033e-32j,
 4.00000000e-01 +9.79685083e-17j,  -3.46370983e-17 +3.08148791e-32
```

上面产生的是周期为4个取样(N/2)的正弦和余弦信号，其FFT的有效成分在下标为2的复数中，其中正弦波的放大系数为2,因此频域虚数部分的值为-1；余弦波的放大系数为0.8，因此其对应的值为0.4。

同频率的正弦波和余弦波通过不同的系数叠加，可以产生同频率的各种相位的余弦波，因此我们可以这样来理解频域中的复数:

- 复数的模（绝对值）代表了此频率的余弦波的振幅
- 复数的辐角代表了此频率的余弦波的相位

让我们来看最后一个例子：

```
>>> x = np.arange(0, 2*np.pi, 2*np.pi/128)
>>> y = 0.3*np.cos(x) + 0.5*np.cos(2*x+np.pi/4) + 0.8*np.cos(3*x-np
>>> yf = np.fft.fft(y)/len(y)
>>> yf[:4]
array([  1.00830802e-17 +0.00000000e+00j,
 1.50000000e-01 +6.27820821e-18j,
 1.76776695e-01 +1.76776695e-01j,   2.00000000e-01 -3.46410162e-01
>>> np.angle(yf[1])
4.1854721366992471e-017
>>> np.abs(yf[1]), np.rad2deg(np.angle(yf[1]))
(0.15000000000000008, 2.3980988870246962e-015)
>>> np.abs(yf[2]), np.rad2deg(np.angle(yf[2]))
(0.25000000000000011, 44.999999999999993)
>>> np.abs(yf[3]), np.rad2deg(np.angle(yf[3]))
(0.39999999999999991, -60.000000000000085)
```

在这个例子中我们产生了三个不同频率的余弦波，并且给他们不同的振幅和相位：

- 周期为128/1.0点的余弦波的相位为0， 振幅为0.3

- 周期为64/2.0点的余弦波的相位为45度， 振幅为0.5
- 周期为128/3.0点的余弦波的相位为-60度，振幅为0.8

对照yf[1], yf[2], yf[3]的复数振幅和辐角，我想你应该对FFT结果中的每个数值所表示的意思有很清楚的理解了吧。

合成时域信号

前面说过通过ifft函数可以将频域信号转换回时域信号，这种转换是精确的。下面我们要写一个小程序，完成类似的事情，不过可以由用户选择只转换一部分频率回到时域信号，这样转换的结果和原始的时域信号会有误差，我们通过观察可以发现使用的频率信息越多，则此误差越小，直观地看到如何通过多个余弦波逐步逼近任意的曲线信号的:

```
# -*- coding: utf-8 -*-
# 本程序演示如何用多个正弦波合成三角波
import numpy as np
import pylab as pl

# 取FFT计算的结果freqs中的前n项进行合成，返回合成结果，计算loops个周期的波形
def fft_combine(freqs, n, loops=1):
    length = len(freqs) * loops
    data = np.zeros(length)
    index = loops * np.arange(0, length, 1.0) / length * (2 * np.pi
    for k, p in enumerate(freqs[:n]):
        if k != 0: p *= 2 # 除去直流成分之外，其余的系数都*2
        data += np.real(p) * np.cos(k*index) # 余弦成分的系数为实数部
        data -= np.imag(p) * np.sin(k*index) # 正弦成分的系数为负的虚数
    return index, data

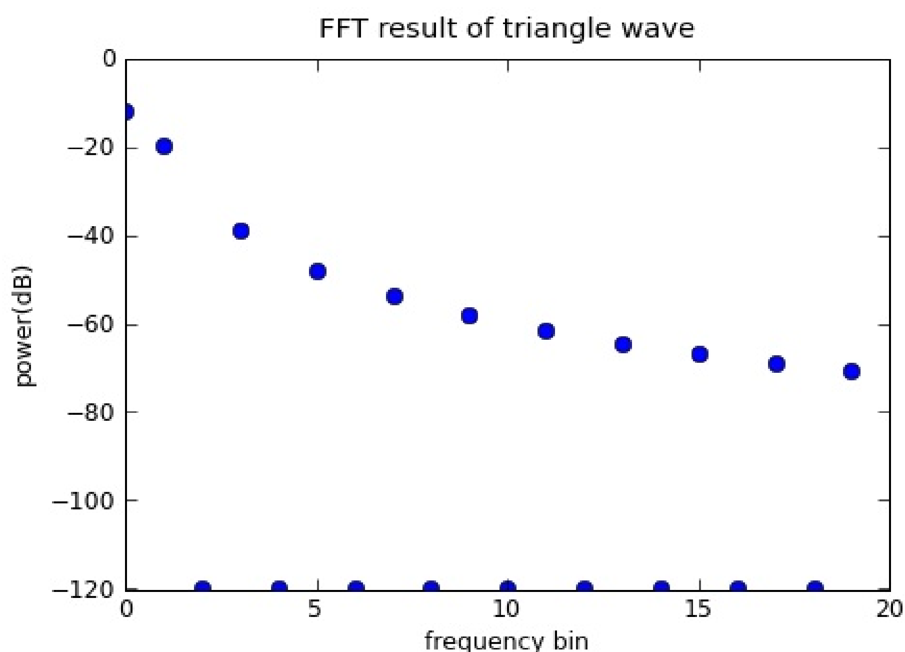
# 产生size点取样的三角波，其周期为1
def triangle_wave(size):
    x = np.arange(0, 1, 1.0/size)
    y = np.where(x<0.5, x, 0)
    y = np.where(x>=0.5, 1-x, y)
    return x, y

fft_size = 256

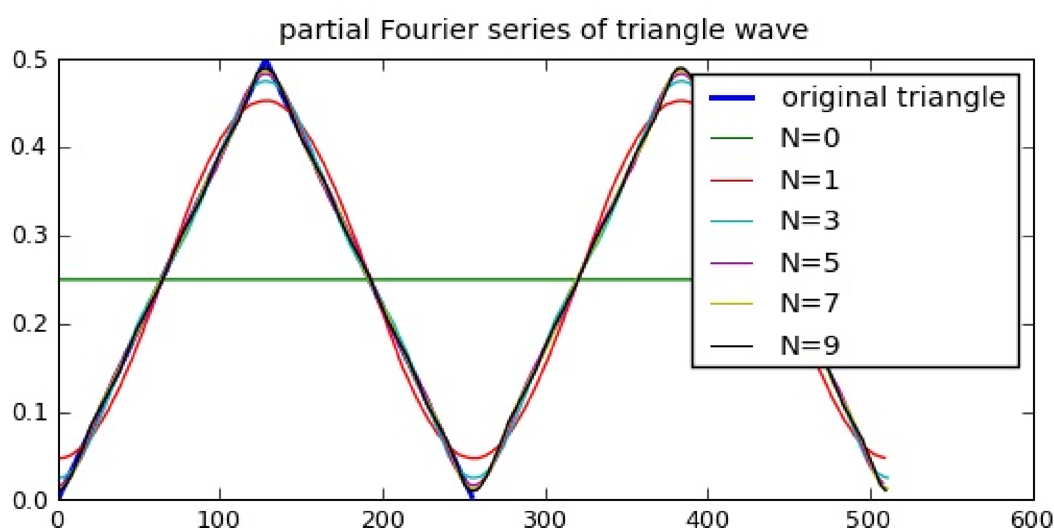
# 计算三角波和其FFT
x, y = triangle_wave(fft_size)
fy = np.fft.fft(y) / fft_size

# 绘制三角波的FFT的前20项的振幅，由于不含下标为偶数的值均为0， 因此取
# log之后无穷小，无法绘图，用np.clip函数设置数组值的上下限，保证绘图正确
pl.figure()
pl.plot(np.clip(20*np.log10(np.abs(fy[:20])), -120, 120), "o")
pl.xlabel("frequency bin")
pl.ylabel("power(dB)")
pl.title("FFT result of triangle wave")

# 绘制原始的三角波和用正弦波逐级合成的结果，使用取样点为X轴坐标
pl.figure()
pl.plot(y, label="original triangle", linewidth=2)
for i in [0,1,3,5,7,9]:
    index, data = fft_combine(fy, i+1, 2)  # 计算两个周期的合成波形
    pl.plot(data, label = "N=%s" % i)
pl.legend()
pl.title("partial Fourier series of triangle wave")
pl.show()
```



三角波的频谱



部分频谱重建的三角波

第18行的triangle_wave函数产生一个周期的三角波形，注意我们使用np.where函数计算区间函数的值。triangle函数返回两个数组，分别表示x轴和y轴的值。注意后面的计算和绘图不使用x轴坐标，而是直接用取样次数作为x轴坐标。

第7行的fft_combine的函数使用fft的结果freqs中的前n个数据重新合成时域信号，由于合成所使用的信号都是正弦波这样的周期信号，所以我们可以通过第三个参数loops指定计算几个周期。

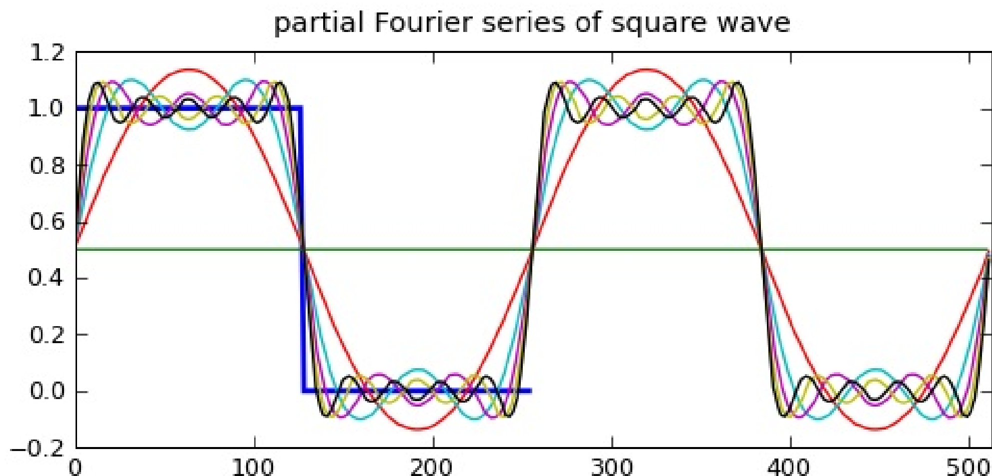
通过这个例子，我们可以看出使用的频率越多，最终合成的波形越接近原始的三角波。

合成方波

由于方波的波形中存在跳变，因此用有限个正弦波合成的方波在跳变出现抖动现象，如下图所示，用正弦波合成的方波的收敛速度比三角波慢得多：

计算方波的波形可以采用如下的函数：

```
def square_wave(size):
    x = np.arange(0, 1, 1.0/size)
    y = np.where(x<0.5, 1.0, 0)
    return x, y
```



正弦波合成方波在跳变处出现都抖动

三角波FFT演示程序

我们希望制作一个用户友好的界面，交互式地观察各种三角波的频谱以及其正弦合成的近似波形。 制作界面是一件很费工的事情，幸好我们有TraitsUI库的帮忙，不到200行程序就可以制作出如下的效果了:



程序中已经给出了详细的注释，相信大家能够读懂并掌握这类程序的写法，其中需要注意的几点:

- 16行，用ScrubberEditor创建一个自定义样式的拖动调整值的控件，77-80行设置Item的editor = scrubber，这样就用我们自定义的控件修改trait属性了，如果不指定editor的话，Range类型的trait属性将以一个滚动条做为编辑器。
- 用Range traits可以指定一个带范围的属性，它可以设置上限下限，上下限可以用整数或者浮点数直接指定，也可以用另外一个trait属性指定(用字符串指定trait属性名)，但是几种类型不能混用，因此程序中专门设计了两个常数的trait属性(36, 37行)，它们的作用只是用来指定其它trait属性的上下限。

  ```
  low = Float(0.02)
  hi = Float(1.0)
  ```

- 190行的triangle_func函数返回一个用frompyfunc创建的ufunc函数，150行用此函数计算三角波，但是用frompyfunc创建的ufunc函数返回的数组的元素的类型(dtype)为object，因此需要用cast["float64"]函数强制将其转换为类型为float64的数组。
  - 处理trait属性的改变事件最简单的方式就是用固定的函数名: _trait属性名_changed，但是当多个trait属性需要共用某一个处理函数时，用@on_trait_change更加简洁。

下面是完整的程序:

```
# -*- coding: utf-8 -*-
from enthought.traits.api import \
    Str, Float, HasTraits, Property, cached_property, Range, Instar

from enthought.chaco.api import Plot, AbstractPlotData, ArrayPlotDa

from enthought.traits.ui.api import \
    Item, View, VGroup, HSplit, ScrubberEditor, VSplit

from enthought.enable.api import Component, ComponentEditor
from enthought.chaco.tools.api import PanTool, ZoomTool

import numpy as np

# 鼠标拖动修改值的控件的样式
scrubber = ScrubberEditor(
    hover_color  = 0xFFFFFF,
    active_color = 0xA0CD9E,
    border_color = 0x808080
)

# 取FFT计算的结果freqs中的前n项进行合成，返回合成结果，计算loops个周期的波形
def fft_combine(freqs, n, loops=1):
    length = len(freqs) * loops
    data = np.zeros(length)
    index = loops * np.arange(0, length, 1.0) / length * (2 * np.pi
    for k, p in enumerate(freqs[:n]):
        if k != 0: p *= 2 # 除去直流成分之外，其余的系数都*2
        data += np.real(p) * np.cos(k*index) # 余弦成分的系数为实数部
        data -= np.imag(p) * np.sin(k*index) # 正弦成分的系数为负的虚数
    return index, data

class TriangleWave(HasTraits):
    # 指定三角波的最窄和最宽范围，由于Range似乎不能将常数和traits名混用
    # 所以定义这两个不变的trait属性
    low = Float(0.02)
    hi = Float(1.0)

    # 三角波形的宽度
    wave_width = Range("low", "hi", 0.5)

    # 三角波的顶点C的x轴坐标
    length_c = Range("low", "wave_width", 0.5)

    # 三角波的定点的y轴坐标
    height_c = Float(1.0)

    # FFT计算所使用的取样点数，这里用一个Enum类型的属性以供用户从列表中选择
    fftsize = Enum( [(2**x) for x in range(6, 12)])

    # FFT频谱图的x轴上限值
    fft_graph_up_limit = Range(0, 400, 20)
```

```
    # 用于显示FFT的结果
    peak_list = Str

    # 采用多少个频率合成三角波
    N = Range(1, 40, 4)

    # 保存绘图数据的对象
    plot_data = Instance(AbstractPlotData)

    # 绘制波形图的容器
    plot_wave = Instance(Component)

    # 绘制FFT频谱图的容器
    plot_fft  = Instance(Component)

    # 包括两个绘图的容器
    container = Instance(Component)

    # 设置用户界面的视图， 注意一定要指定窗口的大小，这样绘图容器才能正常初始化
    view = View(
        HSplit(
            VSplit(
                VGroup(
                    Item("wave_width", editor = scrubber, label=u"波
                    Item("length_c", editor = scrubber, label=u"最高
                    Item("height_c", editor = scrubber, label=u"最高
                    Item("fft_graph_up_limit", editor = scrubber, l
                    Item("fftsize", label=u"FFT点数"),
                    Item("N", label=u"合成波频率数")
                ),
                Item("peak_list", style="custom", show_label=False,
            ),
            VGroup(
                Item("container", editor=ComponentEditor(size=(600,
                orientation = "vertical"
            )
        ),
        resizable = True,
        width = 800,
        height = 600,
        title = u"三角波FFT演示"
    )

    # 创建绘图的辅助函数，创建波形图和频谱图有很多类似的地方，因此单独用一个函数以
    # 减少重复代码
    def _create_plot(self, data, name, type="line"):
        p = Plot(self.plot_data)
        p.plot(data, name=name, title=name, type=type)
        p.tools.append(PanTool(p))
        zoom = ZoomTool(component=p, tool_mode="box", always_on=Fal
        p.overlays.append(zoom)
        p.title = name
```

```
        return p

    def __init__(self):
        # 首先需要调用父类的初始化函数
        super(TriangleWave, self).__init__()

        # 创建绘图数据集，暂时没有数据因此都赋值为空，只是创建几个名字，以供Pl
        self.plot_data = ArrayPlotData(x=[], y=[], f=[], p=[], x2=[

        # 创建一个垂直排列的绘图容器，它将频谱图和波形图上下排列
        self.container = VPlotContainer()

        # 创建波形图，波形图绘制两条曲线： 原始波形(x,y)和合成波形(x2,y2)
        self.plot_wave = self._create_plot(("x","y"), "Triangle Wav
        self.plot_wave.plot(("x2","y2"), color="red")

        # 创建频谱图，使用数据集中的f和p
        self.plot_fft  = self._create_plot(("f","p"), "FFT", type="

        # 将两个绘图容器添加到垂直容器中
        self.container.add( self.plot_wave )
        self.container.add( self.plot_fft )

        # 设置
        self.plot_wave.x_axis.title = "Samples"
        self.plot_fft.x_axis.title = "Frequency pins"
        self.plot_fft.y_axis.title = "(dB)"

        # 改变fftsize为1024，因为Enum的默认缺省值为枚举列表中的第一个值
        self.fftsize = 1024

    # FFT频谱图的x轴上限值的改变事件处理函数，将最新的值赋值给频谱图的响应属性
    def _fft_graph_up_limit_changed(self):
        self.plot_fft.x_axis.mapper.range.high = self.fft_graph_up_

    def _N_changed(self):
        self.plot_sin_combine()

    # 多个trait属性的改变事件处理函数相同时，可以用@on_trait_change指定
    @on_trait_change("wave_width, length_c, height_c, fftsize")
    def update_plot(self):
        # 计算三角波
        global y_data
        x_data = np.arange(0, 1.0, 1.0/self.fftsize)
        func = self.triangle_func()
        # 将func函数的返回值强制转换成float64
        y_data = np.cast["float64"](func(x_data))

        # 计算频谱
        fft_parameters = np.fft.fft(y_data) / len(y_data)

        # 计算各个频率的振幅
        fft_data = np.clip(20*np.log10(np.abs(fft_parameters))[:sel
```

```
        # 将计算的结果写进数据集
        self.plot_data.set_data("x", np.arange(0, self.fftsize)) #
        self.plot_data.set_data("y", y_data)
        self.plot_data.set_data("f", np.arange(0, len(fft_data))) #
        self.plot_data.set_data("p", fft_data)

        # 合成波的x坐标为取样点，显示2个周期
        self.plot_data.set_data("x2", np.arange(0, 2*self.fftsize)

        # 更新频谱图x轴上限
        self._fft_graph_up_limit_changed()

        # 将振幅大于-80dB的频率输出
        peak_index = (fft_data > -80)
        peak_value = fft_data[peak_index][:20]
        result = []
        for f, v in zip(np.flatnonzero(peak_index), peak_value):
            result.append("%s : %s" %(f, v) )
        self.peak_list = "\n".join(result)

        # 保存现在的fft计算结果，并计算正弦合成波
        self.fft_parameters = fft_parameters
        self.plot_sin_combine()

    # 计算正弦合成波，计算2个周期
    def plot_sin_combine(self):
        index, data = fft_combine(self.fft_parameters, self.N, 2)
        self.plot_data.set_data("y2", data)

    # 返回一个ufunc计算指定参数的三角波
    def triangle_func(self):
        c = self.wave_width
        c0 = self.length_c
        hc = self.height_c

        def trifunc(x):
            x = x - int(x) # 三角波的周期为1，因此只取x坐标的小数部分进行
            if x >= c: r = 0.0
            elif x < c0: r = x / c0 * hc
            else: r = (c-x) / (c-c0) * hc
            return r

        # 用trifunc函数创建一个ufunc函数，可以直接对数组进行计算, 不过通过此
        # 计算得到的是一个Object数组，需要进行类型转换
        return np.frompyfunc(trifunc, 1, 1)

if __name__ == "__main__":
    triangle = TriangleWave()
    triangle.configure_traits()
```

频域信号处理

用FFT(快速傅立叶变换)能将时域的数字信号转换为频域信号。转换为频域信号之后我们可以很方便地分析出信号的频率成分，在频域上进行处理，最终还可以将处理完毕的频域信号通过IFFT(逆变换)转换为时域信号，实现许多在时域无法完成的信号处理算法。本章通过几个实例，简单地介绍有关频域信号处理的一些基本知识。

观察信号的频谱

将时域信号通过FFT转换为频域信号之后，将其各个频率分量的幅值绘制成图，可以很直观地观察信号的频谱。下面的程序完成这一任务：

```
# -*- coding: utf-8 -*-
import numpy as np
import pylab as pl

sampling_rate = 8000
fft_size = 512
t = np.arange(0, 1.0, 1.0/sampling_rate)
x = np.sin(2*np.pi*156.25*t)  + 2*np.sin(2*np.pi*234.375*t)
xs = x[:fft_size]
xf = np.fft.rfft(xs)/fft_size
freqs = np.linspace(0, sampling_rate/2, fft_size/2+1)
xfp = 20*np.log10(np.clip(np.abs(xf), 1e-20, 1e100))
pl.figure(figsize=(8,4))
pl.subplot(211)
pl.plot(t[:fft_size], xs)
pl.xlabel(u"时间(秒)")
pl.title(u"156.25Hz和234.375Hz的波形和频谱")
pl.subplot(212)
pl.plot(freqs, xfp)
pl.xlabel(u"频率(Hz)")
pl.subplots_adjust(hspace=0.4)
pl.show()
```

下面逐行对这个程序进行解释：

首先定义了两个常数：sampling_rate, fft_size，分别表示数字信号的取样频率和FFT的长度。

然后调用np.arange产生1秒钟的取样时间，t中的每个数值直接表示取样点的时间，因此其间隔为取样周期1/sampline_rate：

```
t = np.arange(0, 1.0, 1.0/sampling_rate)
```

用取样时间数组t可以很方便地调用函数计算出波形数据，这里计算的是两个正弦波的叠加，一个频率是156.25Hz，一个是234.375Hz：

```
x = np.sin(2*np.pi*156.25*t)  + 2*np.sin(2*np.pi*234.375*t)
```

为什么选择这两个奇怪的频率呢？因为这两个频率的正弦波在512个取样点中正好有整数个周期。满足这个条件波形的FFT结果能够精确地反映其频谱。

N点FFT能精确计算的频率

假设取样频率为fs, 取波形中的N个数据进行FFT变换。那么这N点数据包含整数个周期的波形时，FFT所计算的结果是精确的。于是能精确计算的波形的周期是: n*fs/N。对于8kHz取样，512点FFT来说，8000/512.0 = 15.625Hz，前面的156.25Hz和234.375Hz正好是其10倍和15倍。

下面从波形数据x中截取fft_size个点进行fft计算。np.fft库中提供了一个rfft函数，它方便我们对实数信号进行FFT计算。根据FFT计算公式，为了正确显示波形能量，还需要将rfft函数的结果除以fft_size：

```
xs = x[:fft_size]
xf = np.fft.rfft(xs)/fft_size
```

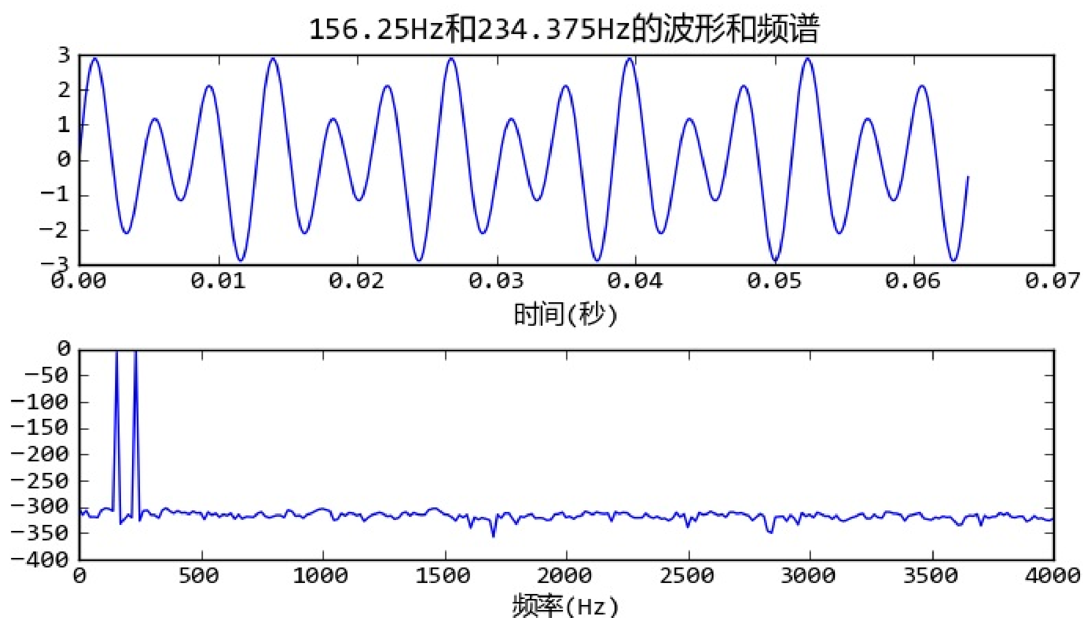
rfft函数的返回值是N/2+1个复数，分别表示从0(Hz)到sampling_rate/2(Hz)的N/2+1点频率的成分。于是可以通过下面的np.linspace计算出返回值中每个下标对应的真正的频率：

```
freqs = np.linspace(0, sampling_rate/2, fft_size/2+1)
```

最后我们计算每个频率分量的幅值，并通过 20*np.log10() 将其转换为以db单位的值。为了防止0幅值的成分造成log10无法计算，我们调用np.clip对xf的幅值进行上下限处理：

```
xfp = 20*np.log10(np.clip(np.abs(xf), 1e-20, 1e100))
```

剩下的程序就是将时域波形和频域波形绘制出来，这里就不再详细叙述了。此程序的输出为：



使用FFT计算正弦波的频谱

如果你放大其频谱中的两个峰值的部分的话，可以看到其值分别为：

```
>>> xfp[10]
-6.0205999132796251
>>> xfp[15]
-9.6432746655328714e-16
```

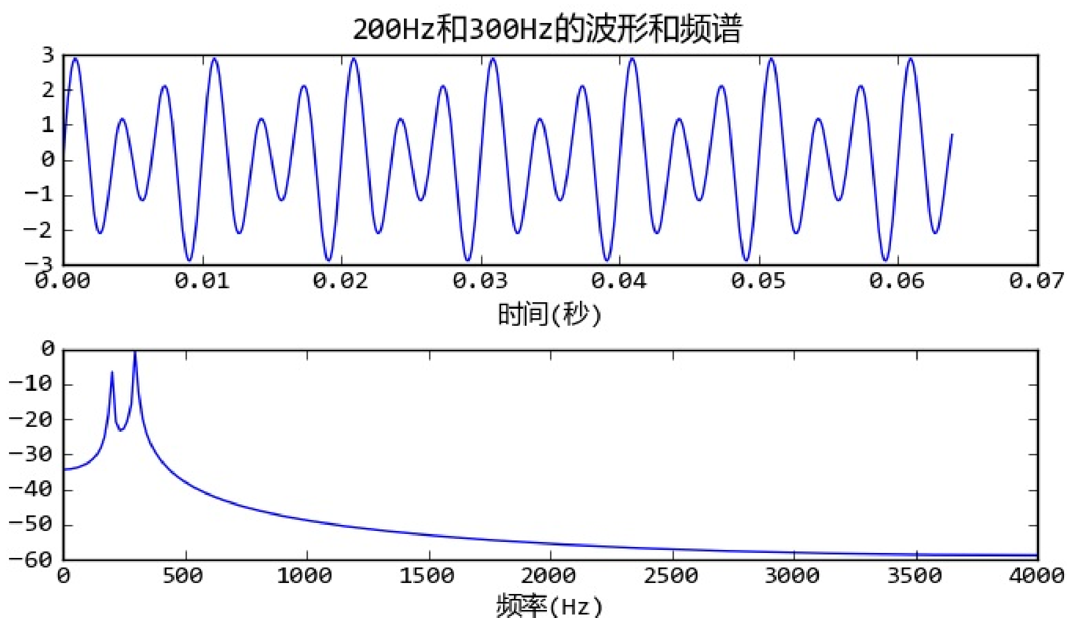
即156.25Hz的成分为-6dB， 而234.375Hz的成分为0dB，与波形的计算公式中的各个分量的能量(振幅值/2)符合。

如果我们波形不能在fft_size个取样中形成整数个周期的话会怎样呢？

将波形计算公式修改为：

```
x = np.sin(2*np.pi*200*t)  + 2*np.sin(2*np.pi*300*t)
```

得到的结果如下：



非完整周期的正弦波经过FFT变换之后出现频谱泄漏

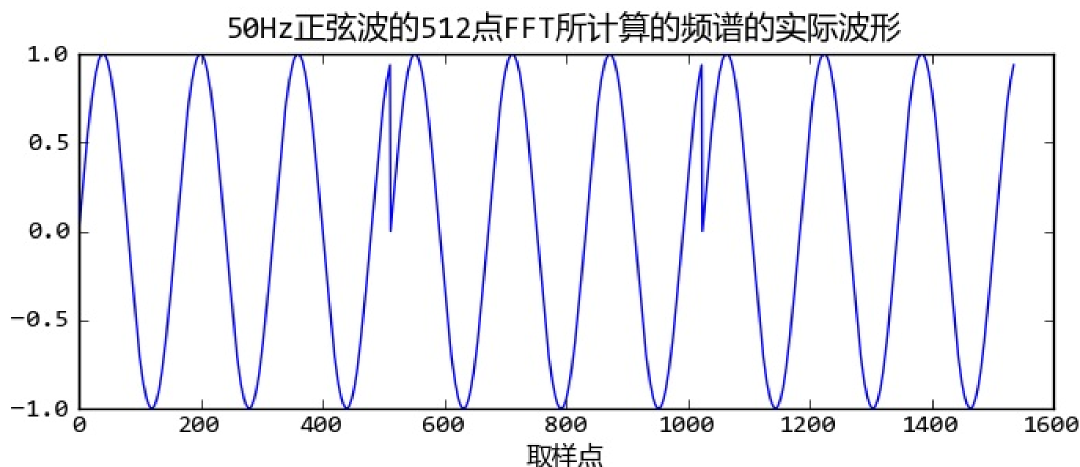
这次得到的频谱不再是两个完美的峰值，而是两个峰值频率周围的频率都有能量。这显然和两个正弦波的叠加波形的频谱有区别。本来应该属于200Hz和300Hz的能量分散到了周围的频率中，这个现象被称为频谱泄漏。出现频谱泄漏的原因在于fft_size个取样点无法放下整数个200Hz和300Hz的波形。

频谱泄漏的解释

我们只能在有限的时间段中对信号进行测量，无法知道在测量范围之外的信号是怎样的。因此只能对测量范围之外的信号进行假设。而傅立叶变换的假设很简单：测量范围之外的信号是所测量到的信号的重复。

现在考虑512点FFT，从信号中取出的512个数据就是FFT的测量范围，它计算的是这512个数据一直重复的波形的频谱。显然如果512个数据包含整数个周期的话，那么得到的结果就是原始信号的频谱，而如果不是整数周期的话，得到的频谱就是如下波形的频谱，这里假设对50Hz的正弦波进行512点FFT：

```
>>> t = np.arange(0, 1.0, 1.0/8000)
>>> x = np.sin(2*np.pi*50*t)[:512]
>>> pl.plot(np.hstack([x,x,x]))
>>> pl.show()
```



50Hz正弦波的512点FFT所计算的频谱的实际波形

由于这个波形的前后不是连续的，出现波形跳变，而跳变处的有着非常广泛的频谱，因此FFT的结果中出现频谱泄漏。

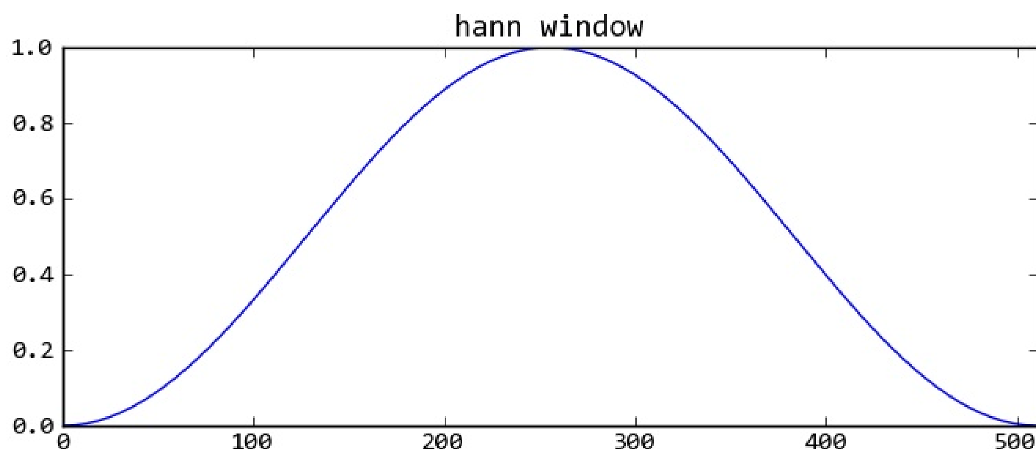
窗函数

为了减少FFT所截取的数据段前后的跳变，可以对数据先乘以一个窗函数，使得其前后数据能平滑过渡。例如常用的hann窗函数的定义如下：

$$w(n) = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$$

其中N为窗函数的点数，下面是一个512点hann窗的曲线：

```
>>> import pylab as pl
>>> import scipy.signal as signal
>>> pl.figure(figsize=(8,3))
>>> pl.plot(signal.hann(512))
```



hann窗函数

窗函数都在scipy.signal库中定义，它们的第一个参数为点数N。可以看出hann窗函数是完全对称的，也就是说第0点和第511点的值完全相同，都为0。在这样的函数和信号数据相乘的话，结果中会出现前后两个连续的0，这样FFT的结果所表示的周期信号中有两个连续的0值，会对信号的周期性有一定的影响。

计算周期信号的一个周期的数据

考虑对一个正弦波取样10个点，那么第一个点的值为0，而最后一个点的值不应该是0，这样这10个数据的重复才能是精确的正弦波，下面的两种计算中，前者是正确的：

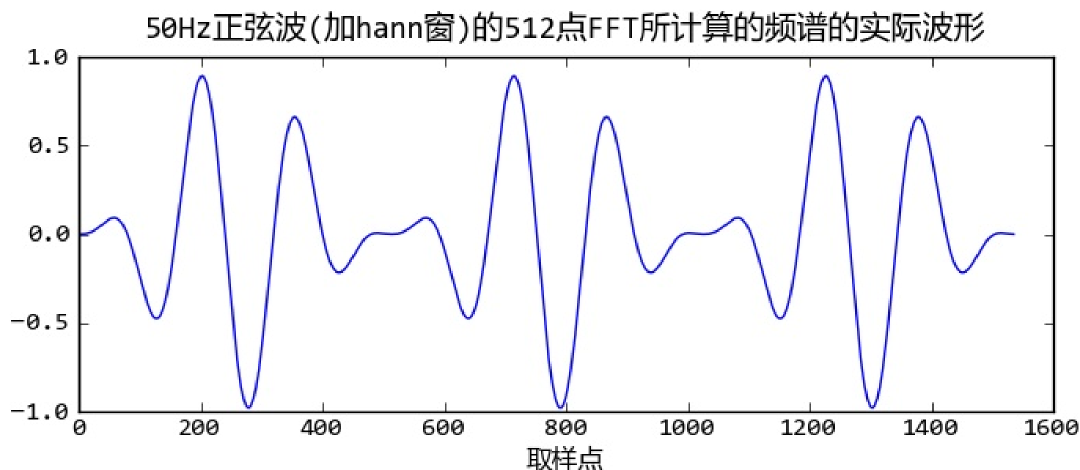
```
>>> np.sin(np.arange(0, 2*np.pi, 2*np.pi/10))
array([  0.00000000e+00,   5.87785252e-01,   9.51056516e-01,
 9.51056516e-01,   5.87785252e-01,   1.22464680e-16,
 -5.87785252e-01,  -9.51056516e-01,  -9.51056516e-01,
 -5.87785252e-01])
>>> np.sin(np.linspace(0, 2*np.pi, 10))
array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
 8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
 -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
 -2.44929360e-16])
```

为了解决连续0值的问题，hann函数提供了一个sym关键字参数，如果设置其为0的话，那么将产生一个N+1点的hann窗函数，然后取其前N个数，这样得到的窗函数适合于周期信号：

```
>>> signal.hann(8)
array([ 0\.         ,  0.1882551 ,  0.61126047,  0.95048443,  0.950
 0.61126047,  0.1882551 ,  0\.        ])
>>> signal.hann(8, sym=0)
array([ 0\.         ,  0.14644661,  0.5        ,  0.85355339,  1\.
 0.85355339,  0.5        ,  0.14644661])
```

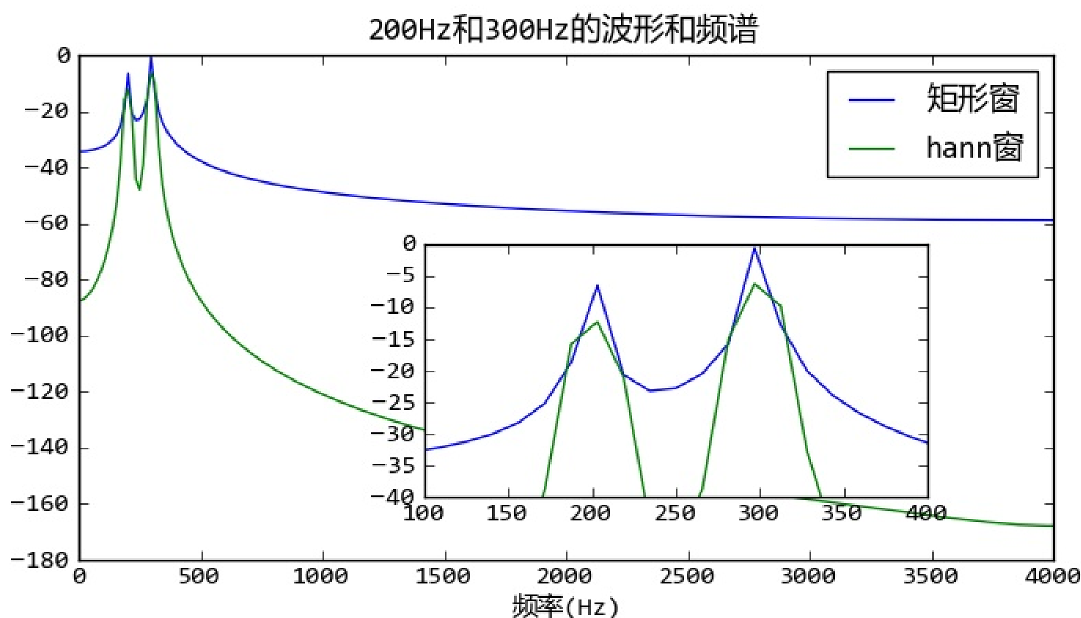
50Hz正弦波与窗函数乘积之后的重复波形如下：

```
>>> t = np.arange(0, 1.0, 1.0/8000)
>>> x = np.sin(2*np.pi*50*t)[:512] * signal.hann(512, sym=0)
>>> pl.plot(np.hstack([x,x,x]))
>>> pl.show()
```



加hann窗的50Hz正弦波的512点FFT所计算的实际波形

回到前面的例子，将200Hz, 300Hz的叠加波形与hann窗乘积之后再计算其频谱，得到如下频谱图：



加hann窗前后的频谱，hann窗能降低频谱泄漏

可以看到与hann窗乘积之后的信号的频谱能量更加集中于200Hz和300Hz，但是其能量有所降低。这是因为hann窗本身有一定的能量衰减：

```
>>> np.sum(signal.hann(512, sym=0))/512
0.5
```

因此如果需要严格保持信号的能量的话，还需要在乘以hann窗之后再乘以2。

上面完整绘图程序请参照：频谱泄漏和*hann*窗

频谱平均

对于频谱特性不随时间变化的信号，例如引擎、压缩机等机器噪声，可以对其进行长时间的采样，然后分段进行FFT计算，最后对每个频率分量的幅值求其平均值可以准确地测量信号的频谱。

下面的程序完成这一计算：

```
import numpy as np
import scipy.signal as signal
import pylab as pl

def average_fft(x, fft_size):
    n = len(x) // fft_size * fft_size
    tmp = x[:n].reshape(-1, fft_size)
    tmp *= signal.hann(fft_size, sym=0)
    xf = np.abs(np.fft.rfft(tmp)/fft_size)
    avgf = np.average(xf, axis=0)
    return 20*np.log10(avgf)
```

average_fft(x, fft_size)对数组x进行fft_size点FFT运算，以dB为单位返回其平均后的幅值。由于x的长度可能不是fft_size的整数倍，因此首先将其缩短为fft_size的整数倍，然后用reshape函数将其转换为一个二维数组tmp。tmp的第1轴的长度为fft_size：

```
n = len(x) // fft_size * fft_size
tmp = x[:n].reshape(-1, fft_size)
```

然后将tmp的第1轴上的数据和窗函数相乘，这里选用的是hann窗：

```
tmp *= signal.hann(fft_size, sym=0)
```

调用rfft对tmp每的行数据进行FFT计算，并求其幅值：

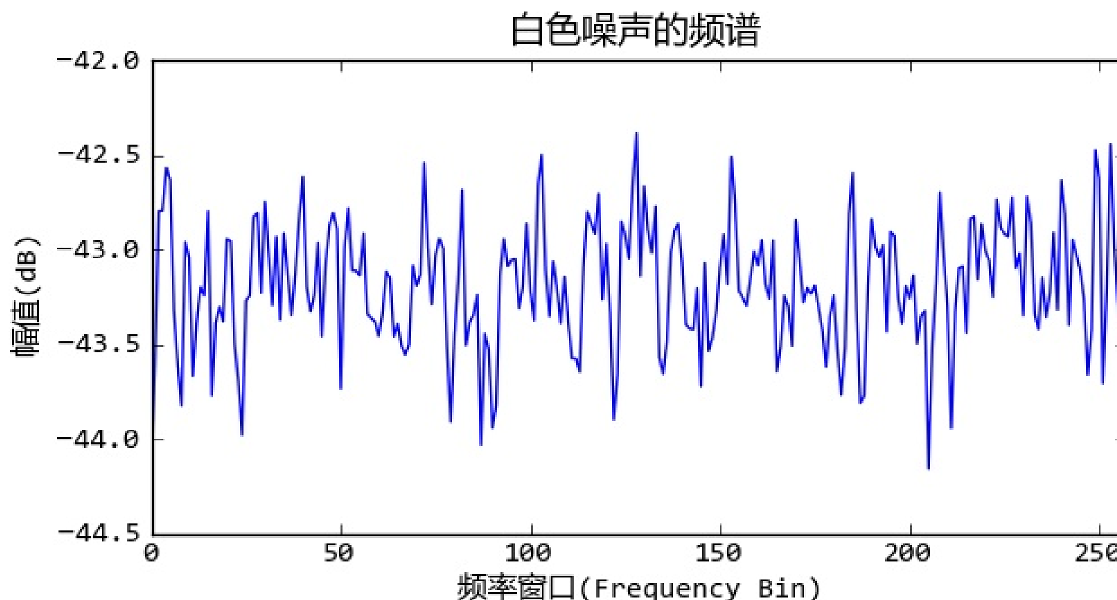
```
xf = np.abs(np.fft.rfft(tmp)/fft_size)
```

接下来调用average函数对xf沿着第0轴进行平均，这样就得到每个频率分量的平均幅值：

```
avgf = np.average(xf, axis=0)
```

下面是利用averagge_fft函数计算随机数序列频谱的例子：

```
>>> x = np.random.rand(100000) - 0.5
>>> xf = average_fft(x, 512)
>>> pl.plot(xf)
>>> pl.show()
```



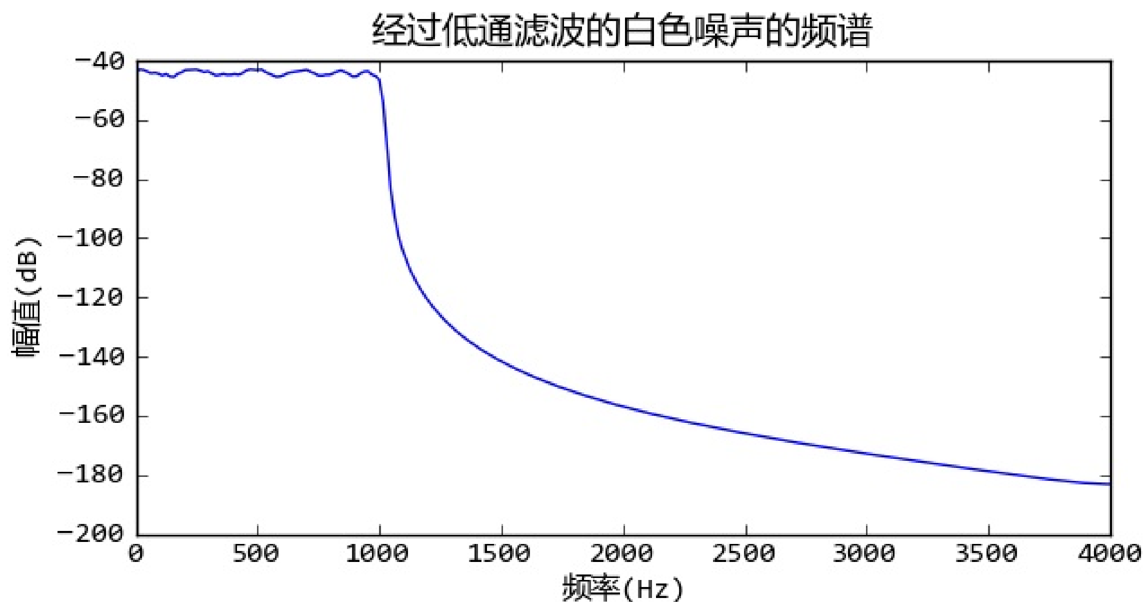
白色噪声的频谱接近水平直线(注意Y轴的范围)

我们可以看到随机噪声的频谱接近一条水平的直线，也就是说每个频率窗口的能量都相同，这种噪声我们称之为白色噪声。

如果我们利用scipy.signal库中的滤波器设计函数，设计一个IIR低通滤波器，将白色噪声输入到此低通滤波器，绘制其输出数据的平均频谱的话，就能够观察到IIR滤波器的频率响应特性，下面的程序利用iirdesign设计一个8kHz取样的1kHz的Chebyshev I 型低通滤波器，iirdesign函数需要用正规化的频率(取值范围为0-1)，然后调用filtfilt对白色噪声信号x进行低通滤波：

```
>>> b,a=signal.iirdesign(1000/4000.0, 1100/4000.0, 1, -40, 0, "chel
>>> x = np.random.rand(100000) - 0.5
>>> y = signal.filtfilt(b, a, x)
```

如果用average_fft计算输出信号y的平均频谱，得到如下频谱图：



经过低通滤波器的白色噪声的频谱

快速卷积

我们知道，信号x经过系统h之后的输出y是x和h的卷积，虽然卷积的计算方法很简单，但是当x和h都很长的时候，卷积计算是非常耗费时间的。因此对于比较长的系统h，需要找到比直接计算卷积更快的方法。

信号系统理论中有这样一个规律：时域的卷积等于频域的乘积，因此要计算时域的卷积，可以将时域信号转换为频域信号，进行乘积运算之后再将结果转换为时域信号，实现快速卷积。

由于FFT运算可以高效地将时域信号转换为频域信号，其运算的复杂度为O(N*log(N))*，因此三次*FFT*运算加一次乘积运算的总复杂度仍然为 *O(N*log(N)) 级别，而卷积运算的复杂度为 O(N*N)，显然通过FFT计算卷积要比直接计算快速得多。这里假设需要卷积的两个信号的长度都为N。

但是有一个问题：FFT运算假设其所计算的信号为周期信号，因此通过上述方法计算出的结果实际上是两个信号的循环卷积，而不是线性卷积。为了用FFT计算线性卷积，需要对信号进行补零扩展，使得其长度长于线性卷积结果的长度。

例如，如果我们要计算数组a和b的卷积，a和b的长度都为128，那么它们的卷积结果的长度为 len(a) + len(b) - 1 = 257。为了用FFT能够计算其线性卷积，需要将a和b都扩展到256。下面的程序演示这个计算过程：

```
# -*- coding: utf-8 -*-
import numpy as np

def fft_convolve(a,b):
    n = len(a)+len(b)-1
    N = 2**(int(np.log2(n))+1)
    A = np.fft.fft(a, N)
    B = np.fft.fft(b, N)
    return np.fft.ifft(A*B)[:n]

if __name__ == "__main__":
    a = np.random.rand(128)
    b = np.random.rand(128)
    c = np.convolve(a,b)

    print np.sum(np.abs(c - fft_convolve(a,b)))
```

此程序的输出为直接卷积和FFT快速卷积的结果之间的误差，大约为5e-12左右。

在这段程序中，a,b的长度为128，其卷积结果c的长度为n=255，我们通过下面的算式找到大于n的最小的2的整数次幂：

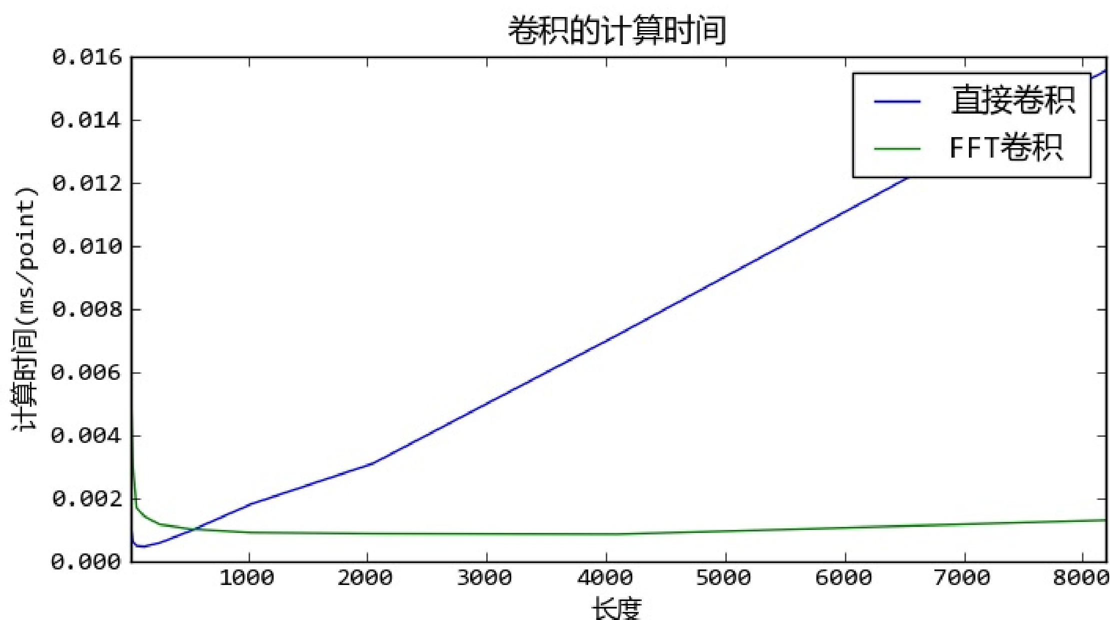
```
N = 2**(int(np.log2(n))+1)
```

在调用fft函数对其进行变换时，传递第二个参数为N(FFT的长度)，这样fft函数将自动对a,b进行补零。最后通过ifft得到的卷积结果c2的长度为N，比实际的卷积结果c要多出一个数，这个多出来的元素应该接近于0，请读者自行验证。

下面测试一下速度：

```
>>> import timeit
>>> setup="""import numpy as np
a=np.random.rand(10000)
b=np.random.rand(10000)
from spectrum_fft_convolve import fft_convolve"""
>>> timeit.timeit("np.convolve(a,b)",setup, number=10)
1.852900578146091
>>> timeit.timeit("fft_convolve(a,b)",setup, number=10)
0.19475575806416145
```

显然计算两个很长的数组的卷积，FFT快速卷积要比直接卷积快很多。但是对于较短的数组，直接卷积运算将更快一些。下图显示了直接卷积和快速卷积的每点的平均计算时间和长度之间的关系：



用FFT计算卷积和直接卷积的时间复杂度比较

由于图中的Y轴表示每点的计算时间，因此对于直接卷积它是线性的：O(N*N)/N。我们看到对于1024点以上的计算，快速卷积显示出明显的优势。

具体的程序请参照 *FFT卷积的速度比较*

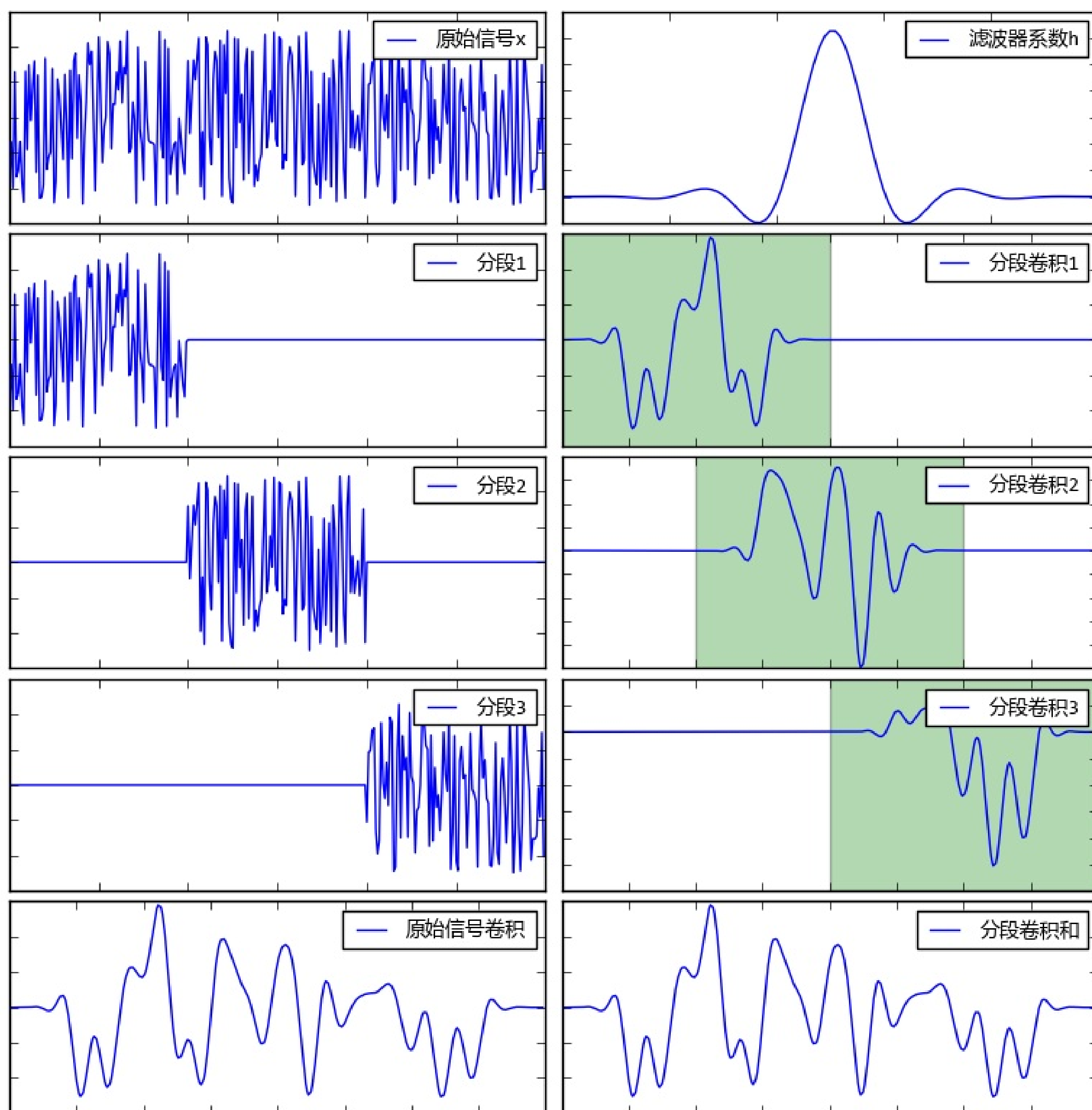
由于FFT卷积很常用，因此scipy.signal库中提供了fftconvolve函数，此函数采用FFT运算可以计算多维数组的卷积。读者也可以参照此函数的源代码帮助理解。

分段运算

现在考虑对于输入信号x和系统响应h的卷积运算，通常x是非常长的，例如要对某段录音进行滤波处理，假设取样频率为8kHz，录音长度为1分钟的话，那么x的长度为480000。而且x的长度也可能不是固定的，例如我们可能需要对麦克风的连续输入信号进行滤波处理。而h的长度通常都是固定的，例如它是某个房间的冲击响应，或者是某种FIR滤波器。

根据前面的介绍，为了有效地利用FFT计算卷积，我们希望它的两个输入长度相当，于是就需要对信号x进行分段处理。对卷积的分段运算被称作：overlap-add运算。

overlap-add的计算方法如下图所示：



分段卷积的过程演示

原始信号x长度为300，将它分为三段，分别与滤波器系数h进行卷积计算，h的长度为101，因此每段输出200个数据，图中用绿色标出每段输出的200个数据。这3段数据按照时间顺序进行求和之后得到结果和原始信号的卷积是相同的。

因此将持续的输入信号x和滤波器h进行卷积的运算可以按照如下步骤进行，假设h的长度为M：

1. 建立一个缓存，其大小为N+M-1，初始值为0
2. 每次从x中读取N个数据，和h进行卷积，得到N+M-1个数据，和缓存中的数据进行求和，并放进缓存中，然后输出缓存前N个数据
3. 将缓存中的数据向左移动N个元素，也就是让缓存中的第N个元素成为第0个元素，后面的N个元素全部设置为0
4. 跳转到2重复运行

下面是实现这一算法的演示程序：

```
# -*- coding: utf-8 -*-
import numpy as np
x = np.random.rand(1000)
h = np.random.rand(101)
y = np.convolve(x, h)

N = 50 # 分段大小
M = len(h) # 滤波器长度

output = []

#缓存初始化为0
buffer = np.zeros(M+N-1,dtype=np.float64)

for i in xrange(len(x)/N):
    #从输入信号中读取N个数据
    xslice = x[i*N:(i+1)*N]
    #计算卷积
    yslice = np.convolve(xslice, h)
    #将卷积的结果加入到缓冲中
    buffer += yslice
    #输出缓存中的前N个数据，注意使用copy，否则输出的是buffer的一个视图
    output.append( buffer[:N].copy() )
    #缓存中的数据左移动N个元素
    buffer[0:M-1] = buffer[N:]
    #后面的补0
    buffer[M-1:] = 0

#将输出的数据组合为数组
y2 = np.hstack(output)
#计算和直接卷积的结果之间的误差
print np.sum(np.abs( y2 - y[:len(x)] ) )
```

注意第23行需要输出缓存前N个数据的拷贝，否则输出的是数组的一个视图，当此后buffer更新时，视图中的数据会一起更新。

将FFT快速卷积和overlap-add相结合，可以制作出一些快速的实时数据滤波算法。但是由于FFT卷积对于两个长度相当的数组时最为有效，因此在分段时也会有所限制：例如如果滤波器的长度为2048，那么理想的分段长度也为2048，如果将分段长度设置得过低，反而会增加运算量。因此在实时性要求很强的系统中，只能采用直接卷积。

Hilbert变换

Hilbert变换能在振幅保持不变的情况下将输入信号的相角偏移90度，简单地说就是能将正弦波形转换为余弦波形，下面的程序验证这一特性：

```
# -*- coding: utf-8 -*-
from scipy import fftpack
import numpy as np
import matplotlib.pyplot as pl

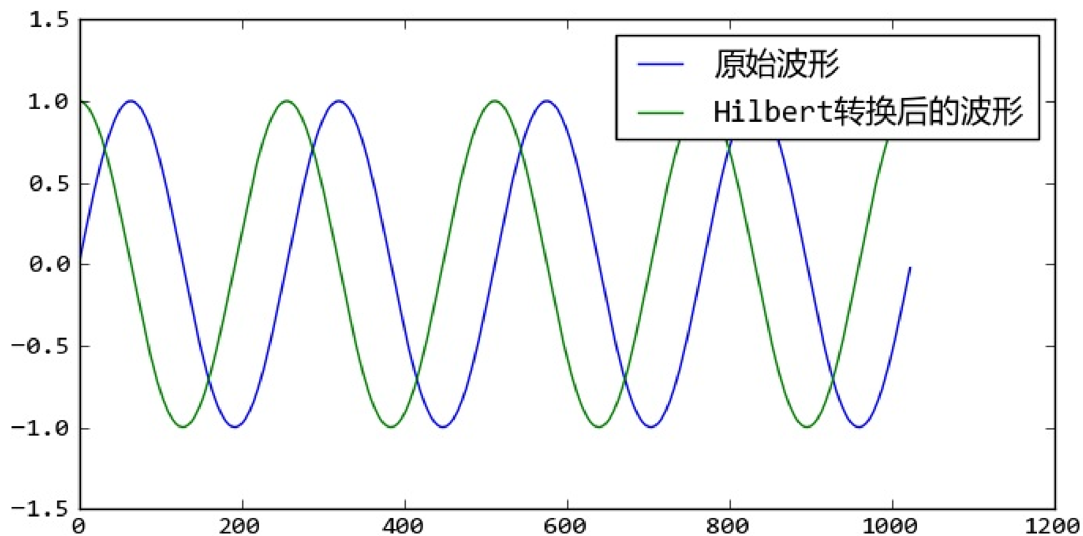
# 产生1024点4个周期的正弦波
t = np.linspace(0, 8*np.pi, 1024, endpoint=False)
x = np.sin(t)

# 进行Hilbert变换
y = fftpack.hilbert(x)
pl.plot(x, label=u"原始波形")
pl.plot(y, label=u"Hilbert转换后的波形")
pl.legend()
pl.show()
```

关于程序的几点说明：

- hilbert转换函数在scipy.fftpack函数库中
- 为了生成完美的正弦波，需要计算整数个周期，因此调用linspace时指定endpoint=False，这样就不会包括区间的结束点：8*np.pi

此程序的输出图表如下，我们可以很清楚地看出hilbert将正弦波变换为了余弦波形。



Hilbert变换将正弦波变为余弦波

Hilbert正变换的相角偏移符号

本书中将相角偏移+90度成为Hilbert正变换。有的文献书籍正好将定义倒转过来：将偏移+90度成为Hilbert负变换，而偏移-90度成为Hilbert正变换。

相角偏移90度相当于复数平面上的点与虚数单位1j相乘，因此Hilbert变换的频率响应可以用如下公式给出：

$H(\omega) = j \cdot sgn(\omega)$

其中 ω 为圆频率，sgn函数为符号函数，即：

$$sgn(\omega) = \begin{cases} 1, & \text{for } \omega > 0, \\ 0, & \text{for } \omega = 0, \\ -1, & \text{for } \omega < 0, \end{cases}$$

我们可以将其频率响应理解为：

- 直流分量为0
- 正频率成分偏移+90度
- 负频率成分偏移-90度

由于对于实数信号来说，正负频率成分共轭，因此对实数信号进行Hilbert变换之后仍然是实数信号。下面的程序验证Hilbert变换的频率响应：

```
>>> x = np.random.rand(16)
>>> y = fftpack.hilbert(x)
>>> X = np.fft.fft(x)
>>> Y = np.fft.fft(y)
>>> np.imag(Y/X)
array([ 0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
  0., -1., -1., -1., -1., -1., -1., -1.])
```

对信号进行N点FFT变换之后：

- 下标为0的频率分量表示直流分量
- 下标为N/2的的频率分量为取样频率/2的频率分量
- 1到N/2-1为正频率分量
- N/2+1到N为负频率分量

对照Y/X的虚数部分，不难看出它是符合Hilbert的频率响应的。如果你用np.real(Y/X)观察实数部分的话，它们全部接近于0。

Hilbert变换可以用作包络检波。具体算法如下所示：

$envelope = \sqrt{H(x)^2 + x^2}$

其中x为原始载波波形，H(x)为x的Hilbert变换之后的波形，envelope为信号x的包络。其原理很容易理解：假设x为正弦波，那么H(x)为余弦波，根据公式：

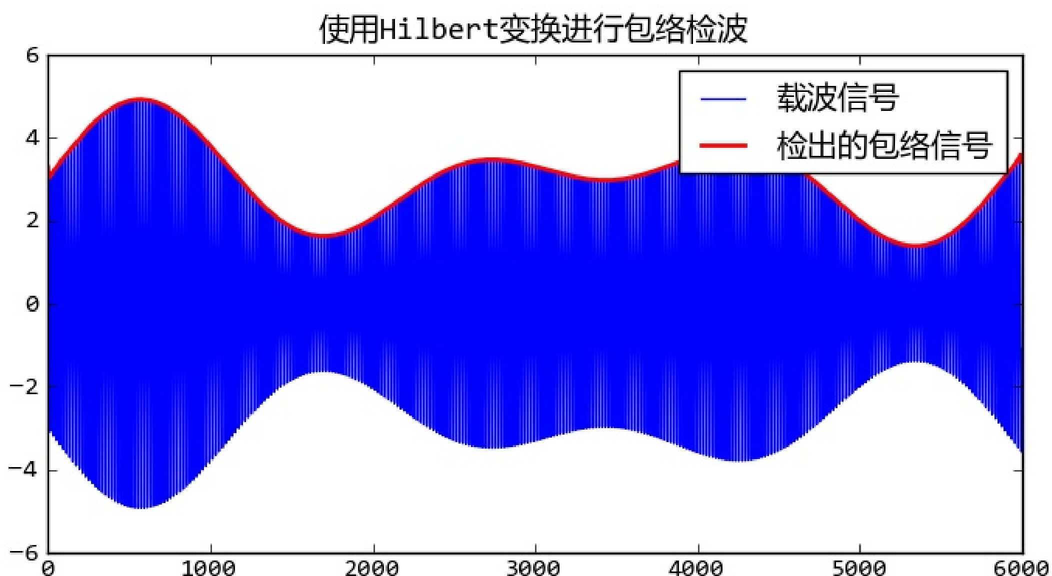
$\sin^2(t) + \cos^2(t) = 1$

可知envelope恒等于1，为sin(t)信号的包络。下面的程序验证这一算法：

```
# -*- coding: utf-8 -*-
import numpy as np
import pylab as pl
from scipy import fftpack

t = np.arange(0, 0.3, 1/20000.0)
x = np.sin(2*np.pi*1000*t) * (np.sin(2*np.pi*10*t) + np.sin(2*np.pi
hx = fftpack.hilbert(x)

pl.plot(x, label=u"载波信号")
pl.plot(np.sqrt(x**2 + hx**2), "r", linewidth=2, label=u"检出的包络信
pl.title(u"使用Hilbert变换进行包络检波")
pl.legend()
pl.show()
```

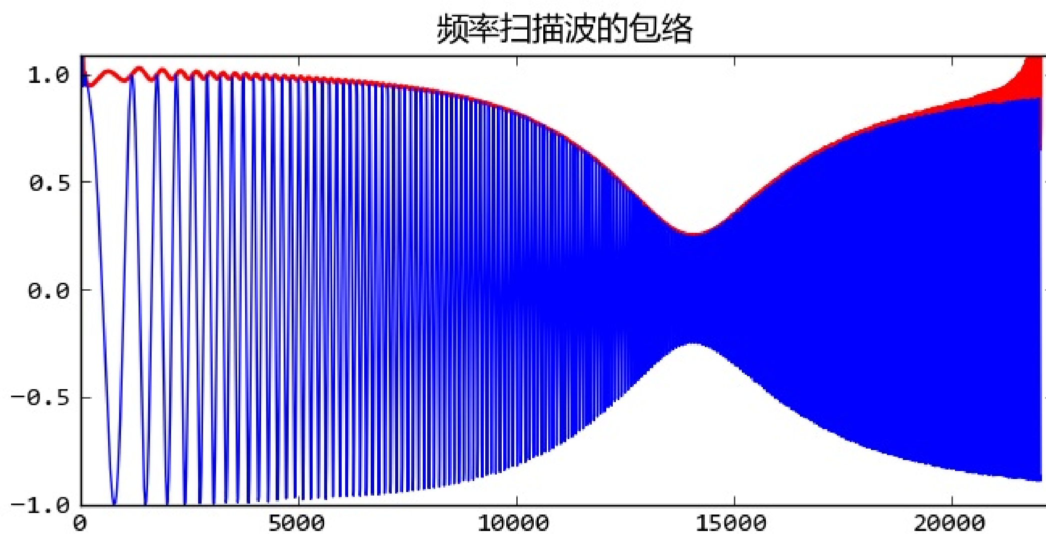


使用Hilbert变换对载波信号进行包络检波

前面介绍过可以使用频率扫描波形测量滤波器的频率响应，我们可以使用这个算法计算出扫描波的包络：

```
>>> run filter_lfilter_example01.py # 运行滤波器的例子
>>> hy = fftpack.hilbert(y)
>>> pl.plot( np.sqrt(y**2 + hy**2),"r", linewidth=2)
>>> pl.plot(y)
>>> pl.title(u"频率扫描波的包络")
>>> pl.show()
```

得到的包络波形如下图所示：



使用Hilbert变换对频率扫描波进行包络检波

可以看出在高频和低频处包络计算出现较大的误差。而在中频部分能很好地计算出包络的形状。

Ctypes和NumPy

用ctypes加速计算

Ctypes是Python处理动态链接库的标准扩展模块，在Windows下使用它可以直接调用C语言编写的DLL动态链接库。由于对传递的参数没有类型和越界检查，因此如果编写的代码有问题的话，很可能会造成程序崩溃。当将数组数据使用指针传递时，出错误的风险将更加大。

为了让程序更加安全，通常会用Python代码对Ctypes调用进行包装，在调用Ctypes之前，在Python级别对数据类型和越界进行检查。这样做会使得调用接口部分比其它的一些手工编写的扩展模块速度要慢，但是如果C语言的代码段处理相当多的数据的话，接口调用部分的速度损失是可以忽略不计的。

用ctypes调用DLL

为了使用CTypes，你必须依次完成以下步骤：

- 编写动态连接库程序
- 载入动态连接库
- 将Python的对象转换为ctypes所能识别的参数
- 使用ctypes的参数调用动态连接库中的函数

下面我们来看看如何用ctypes调用动态链接库。

numpy对ctypes的支持

为了方便动态连接库的载入，numpy提供了一个便捷函数ctypeslib.load_library。它有两个参数，第一个参数是库的文件名，第二个参数是库所在的路径。函数返回的是一个ctypes的对象。通过此对象的属性可以直接到动态连接库所提供的函数。

例如如果我们有一个库名为test_sum.dll，其中提供了一个函数mysum：

```
double mysum(double a[], long n)
{
    double sum = 0;
    int i;
    for(i=0;i<n;i++) sum += a[i];
    return sum;
}
```

的话，我们可以使用如下语句载入此库：

```
>>> from ctypes import *
>>> sum_test = np.ctypeslib.load_library("sum_test", ".")
>>> print sum_test.mysum
<_FuncPtr object at 0x037D7210>
```

要正确调用sum函数，还必须对其参数类型进行说明，下面的语句描述了sum函数的两个参数的类型和返回值的类型进行描述：

```
>>> sum_test.mysum.argtypes = [POINTER(c_double), c_long]
>>> sum_test.mysum.restype = c_double
```

接下来就可以正常调用sum函数了：

```
>>> x = np.arange(1, 101, 1.0)
>>> sum_test.mysum(x.ctypes.data_as(POINTER(c_double)), len(x))
5050.0
```

每次调用sum都需要进行类型转换时比较麻烦的事情，因此可以编写一个Python的mysum函数，将C语言的mysum函数包装起来:

```
def mysum(x):
    return sum_test.mysum(x.ctypes.data_as(POINTER(c_double)), len(
```

在上面的例子中，test_sum.mysum的参数值使用标准的ctypes类型声明：用POINTER(c_double)声明mysum函数的第一个参数是一个指向double的指针；然后调用数组x的x.ctypes.data_as函数将x转换为一个指向double的指针类型。

由于数组的元素在内存中的存储可以是不连续的，而且可以是多维数组，因此我们不能指望前面的mysum函数能够处理所有的情况：

```
>>> x = np.arange(1,11,1.0)
>>>  mysum(x[::2])
15.0
>>> sum(x[::2])
25.0
```

由于x[::2]和x共同一块内存空间，而x[::2]中的元素是不连续的，每个元素之间的间隔为16byptes(2个double的大小)。因此将它传递给mysum的话，实际上计算的是x数组中前5项的和：1+2+3+4+5=15，而实际上我们希望的结果是：1+3+5+7+9=25。

为了对传递的数组参数进行更加详细的描述，numpy库提供了ndpointer函数。ndpointer函数对restype和argtypes中的数组参数进行描述，他有如下4个参数：

- **dtype** : 数组的元素类型
- **ndim** : 数组的维数
- **shape** : 数组的形状，各个轴的长度
- **flags** : 数组的标志

例如：

```
test_sum.mysum.argtypes = [
    np.ctypeslib.ndpointer(dtype=np.float64, ndim=1, flags="C_CONTI
    c_long
]
```

描述了sumfunc函数的参数为一个元素类型为double的、一维的、连续的元素按C语言规定排列的数组。

这时传递给mysum函数的第一个参数可以直接是数组，因此无需再编写一个Python函数对其进行包装：

```
>>> sum_test.mysum(x,len(x))
55.0
>>> sum_test.mysum(x[::2],len(x)/2)
ArgumentError: argument 1: <type 'exceptions.TypeError'>:
array must have flags ['C_CONTIGUOUS']
```

我们注意到如果参数数组不是连续空间的话，mysum函数的调用会抛出异常错误，提醒我们其参数需要C语言排列的连续数组。

如果我们希望它能够处理多维、不连续的数组的话，就需要把数组的shape和strides属性都传递给过去。假设我们想写一个通用的mysum2函数，它可以对二维数组的所有元素进行求和。下面是C语言的程序：

```
double mysum2(double a[], int strides[], int shapes[])
{
    double sum = 0;
    int i, j, M, N, S0, S1;
    M = shape[0]; N=shape[1];
    S0 = strides[0] / sizeof(double);
    S1 = strides[1] / sizeof(double);

    for(i=0;i<M;i++){
        for(j=0;j<N;j++){
            sum += a[i*S0 + j*S1];
        }
    }
    return sum;
}
```

mysum2函数有3个参数，第一个参数a[]指向保存数组数据的内存块；第二个参数astrides指向保存数组各个轴元素之间的间隔(以byte为单位)；第三个参数dims指向保存数组各个轴长度的数组。

由于strides保存的是以byte为单位的间隔长度，因此需要除以sizeof(double)计算出以double为单位的间隔长度S0和S1。这样二维数组a中的第i行、第j列的元素可以通过a[i*S0* + *j*S1]来存取。下面用ctypes对mysum2函数进行包装：

```
sum_test.mysum2.restype = c_double
sum_test.mysum2.argtypes = [
    np.ctypeslib.ndpointer(dtype=np.float64, ndim=2),
    POINTER(c_int),
    POINTER(c_int)
]

def mysum2(x):
    return sum_test.mysum2(x, x.ctypes.strides, x.ctypes.shape)
```

在mysum2函数中，为了将数组x的strides和shape属性传递给C语言的函数，可以使用x.ctypes中提供的strides和shape属性。注意不能直接传递x.strides和x.shape，因为这些是python的tuple对象，而x.ctypes.shape得到的是ctypes包装的整数数组：

```
>>> x = np.zeros((3,4), np.float)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x020B4DF0>
>>> s = x.ctypes.shape
>>> s[0]
3
>>> s[1]
4
```

可以看出x.ctypes.shape是一个有两个元素的C语言长整型数组。虽然我们也可以在Python中通过下标读取其各个元素的值，但是通常它们是作为参数传递给C语言函数用的。

自适应滤波器和NLMS模拟

本章将简要介绍自适应滤波器的原理以及其最常用的算法NLMS，并给NLMS算法的两种实现方法：用纯Python编写，和用ctypes调用C语言编写。最后将对NLMS算法进行一些的实验。

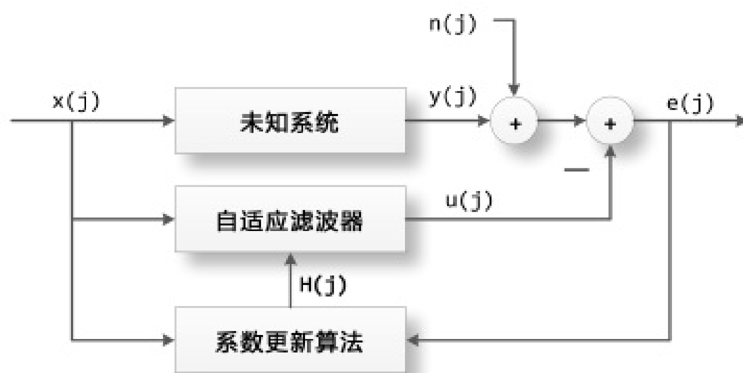
自适应滤波器简介

近年来，随着数字信号处理器的功能的不断增强，自适应信号处理 (adaptive signal process)活跃在噪声消除、回声控制、信号预测、声音定位等众多信号处理领域。

尽管其应用领域十分广泛，但基本的系统构造大致只有如下几种分类。

系统辨识

所谓系统辨识(system identification)，就是通过对未知系统的输入输出进行观测，构造一个滤波器使得它在同样的输入的情况下，输出信号和未知系统相同。简而言之，就是通过观测未知系统对输入的反应，探知其内部情况。为了探知内情而使用的输入信号我们称之为参照信号。



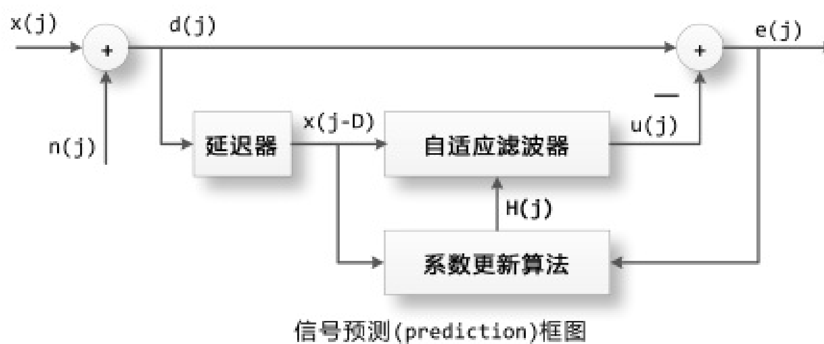
系统识别(System Identification)的框图

如上图所示参照信号 x(j)同时输入到未知系统和自适应滤波器H中，未知系统的输出为y(j)， 自适应滤波器的输出为u(j)，由于观测误差或者外部噪声的干扰，实际观测到的未知系统的输出为d(j)=y(j)+n(j)，n(j)被称为外部干扰。通过求的d(j)和u(j)之间的误差e(j)=d(j)-u(j)，我们可以知道自适应滤波器H和未知系统还有多少差别，通过这个误差我们更新H的内部参数，使得它更加靠近未知系统。

上面各个公式中的j表示某一时刻，因为我们讨论的是数字信号处理，已经对所有的信号进行取样，因此可以把j简单的看作取样点的下标。

信号预测

所谓信号预测就是通过信号过去的值预测（计算）现在的值，下面是信号预测的系统框图。



信号预测(prediction)框图

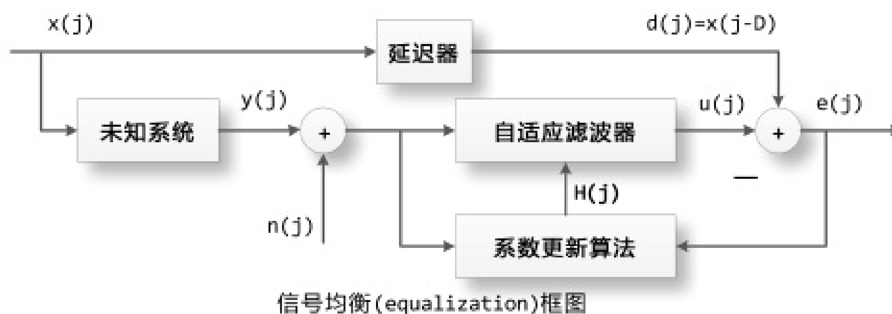
信号预测(Predication)框图

x(j)是待预测的信号，假设我们无法完美地观测此信号，因此导入一个外部干扰n(j)，这样d(j)=x(j)+n(j)就是我们观测到的待预测信号。

通过延迟器将d(j)进行延时得到d(j-D)，并把d(j-D)输入到自适应滤波器H中，得到其输出为u(j)，u(j)就是自适应滤波器通过待预测信号过去的值预测出的现在的值，计算观测值d(j)和预测值u(j)之间的误差e(j)=d(j)-u(j)，通过e(j)更新自适应滤波器H的内部系数使得其输出更加接近d(j)。

如果x(j)存在白色噪声的成分和周期信号的成分，由于白色噪声是完全不自相关，无法预测的信号，因此通过过去的值x(j-D)所能预测的只能是其中的周期信号的成分。这样自适应滤波器H的输出信号u(j)就会与周期信号成分渐渐逼近，而e(j)则是剩下的不可预测的白色噪声的成分。因此自适应滤波器也可以运用于噪声消除。

信号均衡



信号均衡(equalization)框图

信号均衡(Equalization)框图

当信号x(j)通过未知系统之后变成y(j)，未知系统对信号x(j)进行了某种改变，使得其波形产生歪曲。我们希望均衡器矫正这种歪曲，也就是通过y(j)重建原始信号x(j)，由于因果律还原原始信号x(j)是不可能的，我们只能还原其延时了的信号x(j-D)。x(j)和x(j-D)除了时间上的延迟之外，其它特性完全相同。

这里我们将观测到的未知系统的输出y(j)+n(j)输入到自适应滤波器H中，通过H的系数更新使得其输出u(j)逐渐逼近原始信号的延时x(j-D)。这样我们就构建了一个滤波器H使得它与未知系统的卷积正好等于一个脉冲传递函数。也就是说H的频域特性恰好能抵消未知系统的所带来的改变。

NLMS计算公式

自适应滤波器中最重要的一个环节就是其系数的更新算法，如果不对自适应滤波器的系数更新的话，那么它就只是一个普通的FIR滤波器了。系数更新算法有很多种类，最基本、常用、简单的一种方法叫做NLMS(归一化最小均方)，让我们先来看看它的数学公式表达：

设置自适应滤波器系数 $\mathbf{h}$ 的所有初始值为0, $\mathbf{h}$ 的长度为I。

$$\mathbf{h}(0) = 0$$

对每个取样值进行如下计算，其中n=0, 1, 2, ...

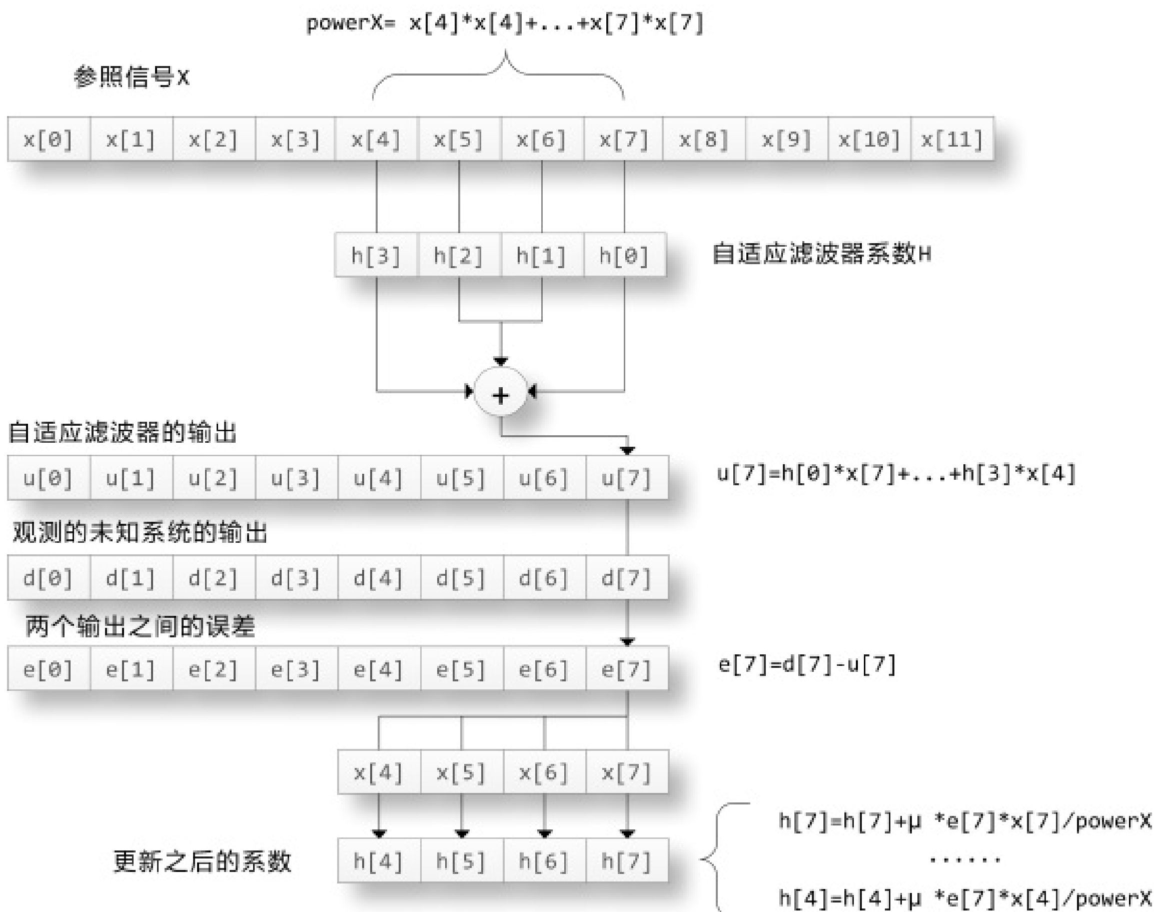
$$\mathbf{x}(n) = [x(n), x(n-1), \dots, x(n-I+1)]^T$$

$$e(n) = d(n) - \mathbf{h}^H(n)\mathbf{x}(n)$$

$$\mathbf{h}(n+1) = \mathbf{h}(n) + \frac{\mu e(n)\mathbf{x}(n)}{\mathbf{x}^H(n)\mathbf{x}(n)}$$

自适应滤波器系数 $\mathbf{h}$ 是一个长度为I的矢量，也就是一个长度为I的FIR滤波器。在时刻n，滤波器的每个系数对应的输入信号为 $\mathbf{x}(n)$，它也是一个长度为I的矢量。这两个矢量的点乘即为滤波器的输出和目标信号d(n)之间的差为e(n)，然后根据e(n)和 $\mathbf{x}(n)$，更新滤波器的系数。

数学公式总是令人难以理解的，下面我们以图示为例进行说明：



NLMS算法示意图

图中假设自适应滤波器h的长度为4，在时刻7滤波器的输出为:

```
u[7] = h[0]*x[7] + h[1]*x[6] + h[2]*x[5] + h[3]*x[4]
```

滤波器的输入信号的平方和powerX为:

```
powerX = x[4]*x[4] + x[5]*x[5] + x[6]*x[6] + x[7]*x[7]
```

未知系统的输出d[7]和滤波器的输出u[7]之间的差为:

```
e[7] = d[7] - u[7]
```

使用u[7]和x[4]..x[7]对滤波器的系数更新:

```
h[4] = h[4] + u * e[7]*x[4]/powerX
h[4] = h[5] + u * e[7]*x[5]/powerX
h[4] = h[6] + u * e[7]*x[6]/powerX
h[4] = h[7] + u * e[7]*x[7]/powerX
```

其中参数u成为更新系数，为0到1之间的一个实数，此值越大系数更新的速度越快。对于每个时刻i都需要进行上述的计算，因此滤波器的系数对于每个参照信号x的取样都更新一次。

NumPy实现

按照上面介绍的NLMS算法，我们很容易写出用NumPy实现的NLMS计算程序：

```
# -*- coding: utf-8 -*-
# filename: nlms_numpy.py

import numpy as np

# 用Numpy实现的NLMS算法
# x为参照信号，d为目标信号，h为自适应滤波器的初值
# step_size为更新系数
def nlms(x, d, h, step_size=0.5):
    i = len(h)
    size = len(x)
    # 计算输入到h中的参照信号的乘方he
    power = np.sum( x[i:i-len(h):-1] * x[i:i-len(h):-1] )
    u = np.zeros(size, dtype=np.float64)

    while True:
        x_input = x[i:i-len(h):-1]
        u[i] = np.dot(x_input , h)
        e = d[i] - u[i]
        h += step_size * e / power * x_input

        power -= x_input[-1] * x_input[-1] # 减去最早的取样
        i+=1
        if i >= size: return u
        power += x[i] * x[i] # 增加最新的取样
```

为了节省计算时间，我们用一个临时变量power保存输入到滤波器h中的参照信号x的能量。在对于x中的每个取样的循环中，power减去x中最早的一个取样值的乘方，增加最新的一个取样值的乘方。这样为了计算参照信号的能量，每次循环只需要计算两次乘法和两次加法即可。

nlms函数的输入为参照信号x、目标信号d和自适应滤波器的系数h。因为在后面的模拟计算中，d是x和未知系统的脉冲响应的卷积而计算的来，它的长度会大于x的参数，因此循环体的循环次数以参照信号的长度为基准。

为了对自适应滤波器的各种应用进行模拟，我们还需要如下的几个辅助函数，完整的程序请参考 *NLMS算法的模拟测试* 。

```
def make_path(delay, length):
    path_length = length - delay
    h = np.zeros(length, np.float64)
    h[delay:] = np.random.standard_normal(path_length) * np.exp( np
    h /= np.sqrt(np.sum(h*h))
    return h
```

make_path产生一个长度为length，最小延时为delay的指数衰减的波形。这种波形和封闭空间的声音的传递函数有些类似之处，因此在计算机上进行声音的算法模拟时经常用这种波形作为系统的传递函数。：

```
def plot_converge(y, u, label=""):
    size = len(u)
    avg_number = 200
    e = np.power(y[:size] - u, 2)
    tmp = e[:int(size/avg_number)*avg_number]
    tmp.shape = -1, avg_number
    avg = np.average( tmp, axis=1 )
    pl.plot(np.linspace(0, size, len(avg)), 10*np.log10(avg), linew

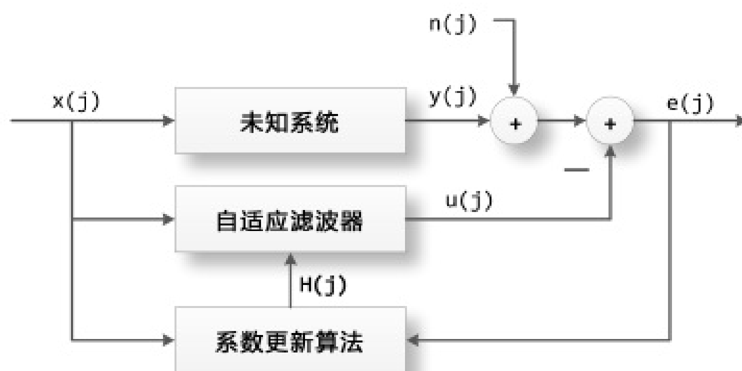
def diff_db(h0, h):
   return 10*np.log10(np.sum((h0-h)*(h0-h)) / np.sum(h0*h0))
```

plot_converge绘制信号y和信号u之间的误差，每avg_number个取样点就上一次误差的乘方的平均值。我们将用plot_converge函数绘制未知系统的输出y和自适应滤波器的输出u之间的误差。观察自适应滤波器是如何收敛的，以评价自适应滤波器的收敛特性。diff_db函数同样是用来评价自适应滤波器的收敛特性，不过他是直接计算未知系统的传递函数h0和自适应滤波器的传递函数h之间的误差。下面我们会看到这两个函数得到的收敛值是相同的。

系统辨识模拟

我们用下面的函数调用nlms算法对系统辨识应用进行模拟：

```
def sim_system_identify(nlms, x, h0, step_size, noise_scale):
       y = np.convolve(x, h0)
       d = y + np.random.standard_normal(len(y)) * noise_scale # 添
       h = np.zeros(len(h0), np.float64) # 自适应滤波器的长度和未知系统
       u = nlms( x, d, h, step_size )
       return y, u, h
```



系统识别(system identification)框图

系统识别(System Identification)的框图

此函数的参数分别为：

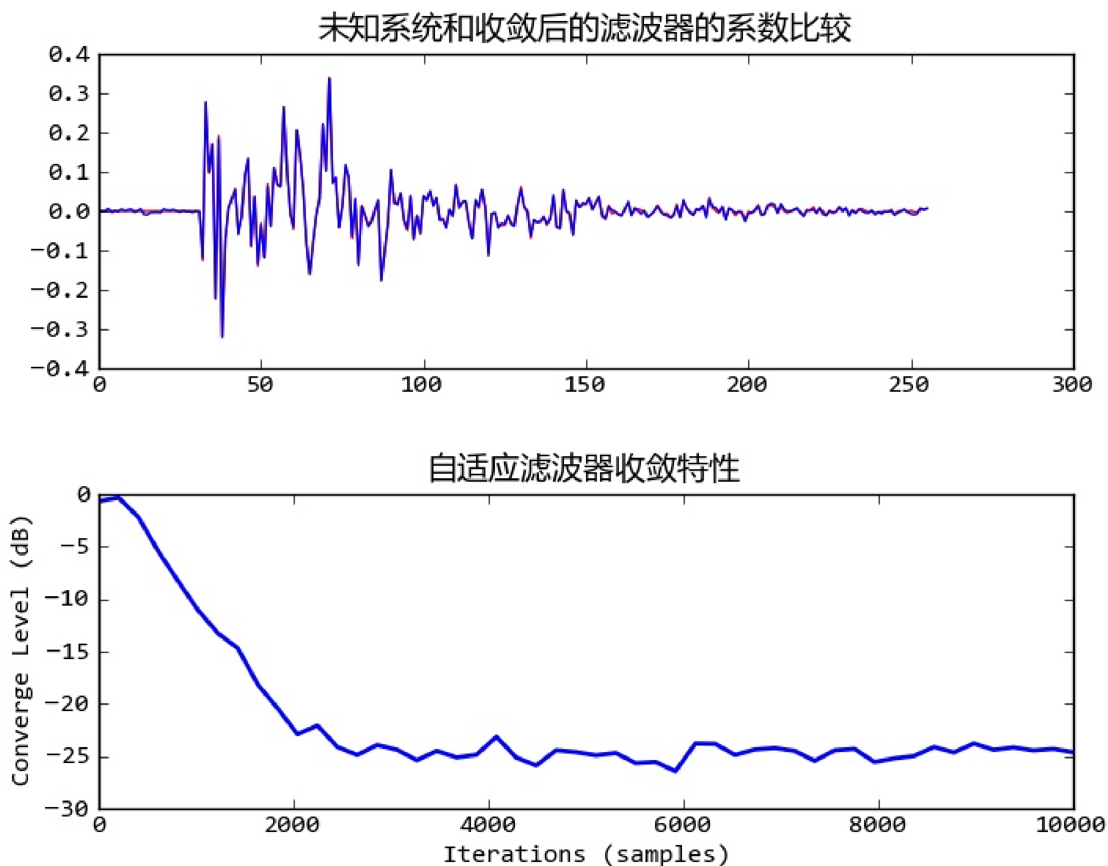
- **nlms** : nlms算法的实现函数
- **x** : 参照信号
- **h0** : 未知系统的传递函数，虽然是未知系统，但是计算机模拟时它是已知的
- **step_size** : nlms算法的更新系数
- **noise_scale** : 外部干扰的系数，此系数决定外部干扰的大小，0表示没有外部干扰

函数的返回值分别为：

- **y** : 未知系统的输出，不包括外部干扰
- **u** : 自适应滤波器的输出
- **h** : 自适应滤波器的最终的系数

最后我们用下面的函数创建未知系统h0， 参照信号x，然后调用sim_system_identify函数得到结果并且绘图：

```
def system_identify_test1():
    h0 = make_path(32, 256) # 随机产生一个未知系统的传递函数
    x = np.random.standard_normal(10000)  # 参照信号为白噪声
    y, u, h = sim_system_identify(nlms_numpy.nlms, x, h0, 0.5, 0.1)
    print diff_db(h0, h)
    pl.figure( figsize=(8, 6) )
    pl.subplot(211)
    pl.subplots_adjust(hspace=0.4)
    pl.plot(h0, c="r")
    pl.plot(h, c="b")
    pl.title(u"未知系统和收敛后的滤波器的系数比较")
    pl.subplot(212)
    plot_converge(y, u)
    pl.title(u"自适应滤波器收敛特性")
    pl.xlabel("Iterations (samples)")
    pl.ylabel("Converge Level (dB)")
    pl.show()
```

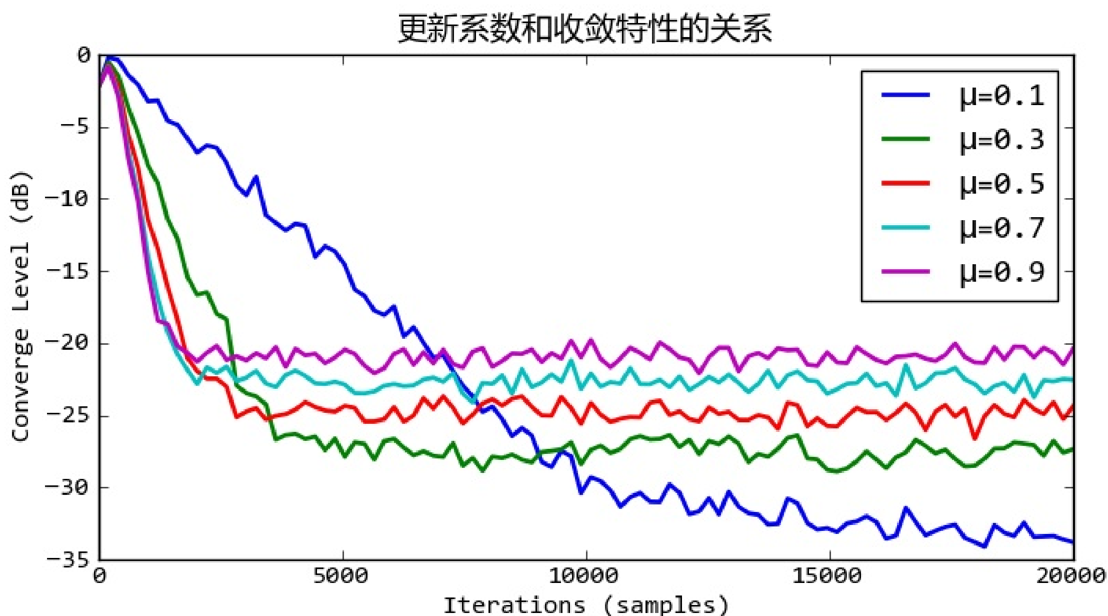


自适应滤波器收敛之后的系数和收敛速度

上部的图显示的是未知系统(红色)和自适应滤波器(蓝色)的传递函数的系数，我们看到自适应滤波器已经十分接近未知系统了。diff_db(h0, h)的输出为-25.35dB。下部的图通过绘制y和u之间的误差，显示了自适应滤波器的收敛过程。我们看到经过约3000点的计算之后，收敛过程已经饱和，最终的误差为-25dB左右，和diff_db计算的结果一致。

从图中可以看到收敛过程的两个重要特性：收敛时间和收敛精度。参照信号的特性、外部干扰的大小和更新系数都会影响这两个特性。下面让我们看看参照信号为白色噪声、外部干扰的能量固定时，更新系数对它们影响：

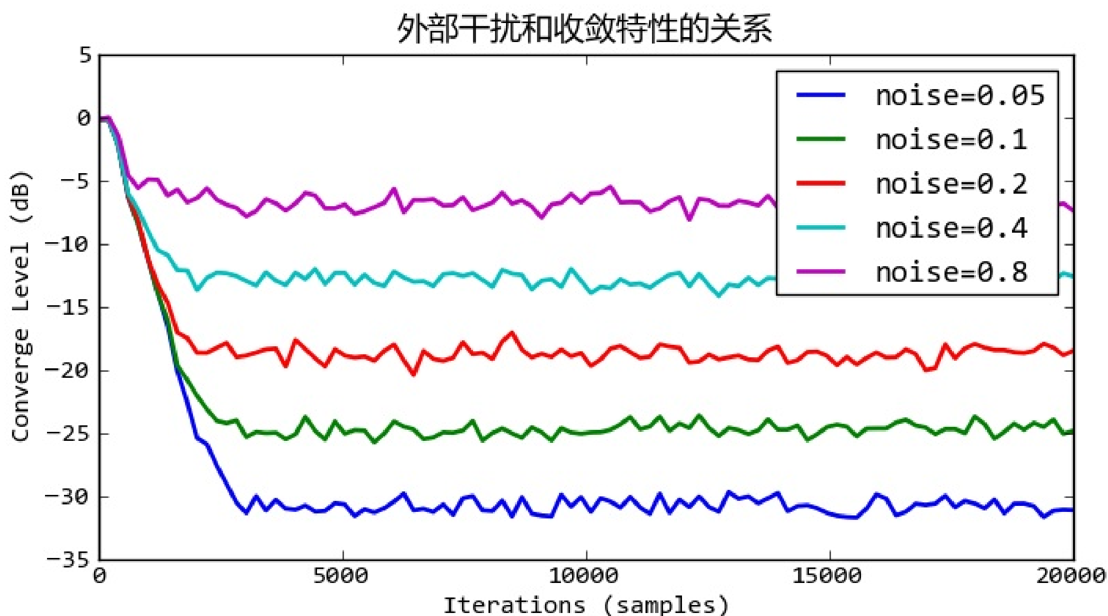
```
def system_identify_test2():
    h0 = make_path(32, 256) # 随机产生一个未知系统的传递函数
    x = np.random.standard_normal(20000)  # 参照信号为白噪声
    pl.figure(figsize=(8,4))
    for step_size in np.arange(0.1, 1.0, 0.2):
        y, u, h = sim_system_identify(nlms_numpy.nlms, x, h0, step_
        plot_converge(y, u, label=u"μ=%s" % step_size)
    pl.title(u"更新系数和收敛特性的关系")
    pl.xlabel("Iterations (samples)")
    pl.ylabel("Converge Level (dB)")
    pl.legend()
    pl.show()
```



更新系数和收敛速度的关系

下面是更新系数固定，外部干扰能量变化时的收敛特性：

```
def system_identify_test3():
    h0 = make_path(32, 256) # 随机产生一个未知系统的传递函数
    x = np.random.standard_normal(20000)  # 参照信号为白噪声
    pl.figure(figsize=(8,4))
    for noise_scale in [0.05, 0.1, 0.2, 0.4, 0.8]:
        y, u, h = sim_system_identify(nlms_numpy.nlms, x, h0, 0.5,
        plot_converge(y, u, label=u"noise=%s" % noise_scale)
    pl.title(u"外部干扰和收敛特性的关系")
    pl.xlabel("Iterations (samples)")
    pl.ylabel("Converge Level (dB)")
    pl.legend()
    pl.show()
```



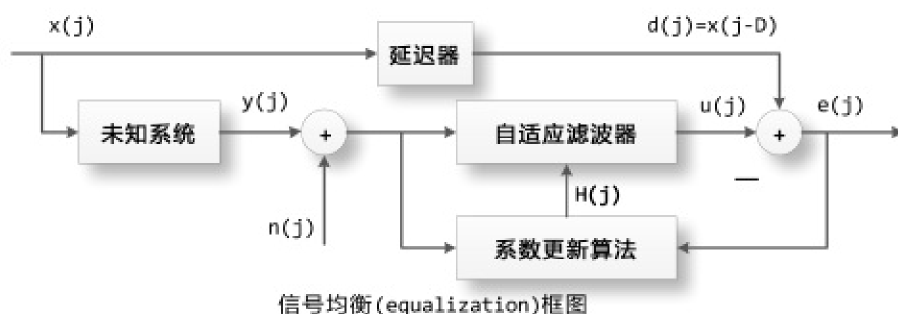
外部干扰噪声和收敛速度的关系

从上面的图可以看出，当外部干扰的振幅增加一倍、能能量增加6dB时，收敛精度降低6dB。而由于更新系数相同，所以收敛过程中的收敛速度都是一样的。

信号均衡模拟

对于信号均衡的应用我们用如下的程序进行模拟：

```
def sim_signal_equation(nlms, x, h0, D, step_size, noise_scale):
    d = x[:-D]
    x = x[D:]
    y = np.convolve(x, h0)[:len(x)]
    h = np.zeros(2*len(h0)+2*D, np.float64)
    y += np.random.standard_normal(len(y)) * noise_scale
    u = nlms(y, d, h, step_size)
    return h
```



信号均衡(Equalization)框图

sim_signal_equation函数的参数：

- **nlms**：nlms算法的实现函数

- **x** : 未知系统的输入信号
- **h0** : 未知系统的传递函数
- **D** : 延迟器的延时参数
- **step_size** : nlms算法的更新系数
- **noise_scale** : 外部干扰的系数，此系数决定外部干扰的大小，0表示没有外部干扰

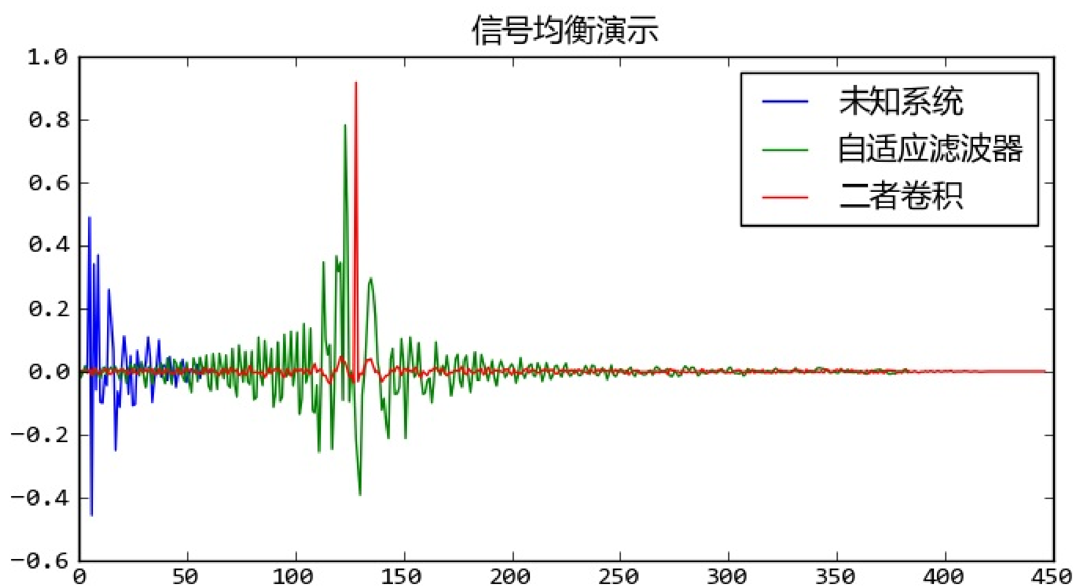
在函数中的各个局部变量：

- **d** : 输入信号经过延迟器之后的信号
- **y** : 未知系统的输出
- **h** : 自适应滤波器的系数，它的长度要足够长，程序中使用 2倍延时 + 2倍未知系统的传递函数的长度

函数的返回值为自适应滤波器收敛后的系数，它能够均衡h0对输入信号所造成的影响。我们通过下面的函数产生数据、调用模拟函数以及绘制结果：

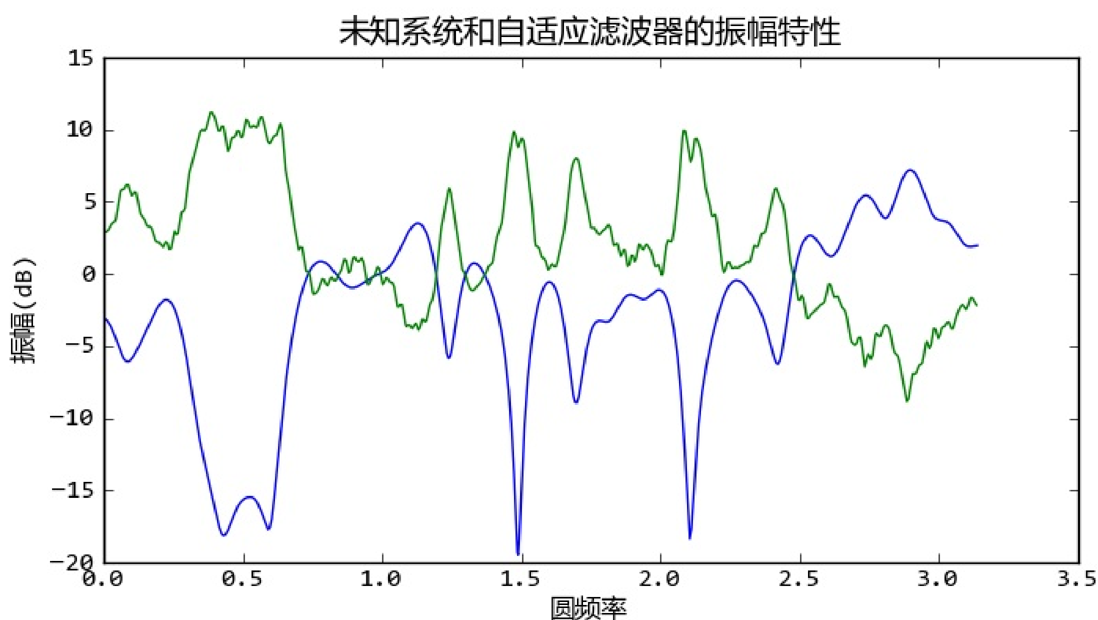
```
def signal_equation_test1():
    h0 = make_path(5, 64)
    D = 128
    length = 20000
    data = np.random.standard_normal(length+D)
    h = sim_signal_equation(nlms_numpy.nlms, data, h0, D, 0.5, 0.1)
    pl.figure(figsize=(8,4))
    pl.plot(h0, label=u"未知系统")
    pl.plot(h, label=u"自适应滤波器")
    pl.plot(np.convolve(h0, h), label=u"二者卷积")
    pl.title(u"信号均衡演示")
    pl.legend()
    w0, H0 = scipy.signal.freqz(h0, worN = 1000)
    w, H = scipy.signal.freqz(h, worN = 1000)
    pl.figure(figsize=(8,4))
    pl.plot(w0, 20*np.log10(np.abs(H0)), w, 20*np.log10(np.abs(H)))
    pl.title(u"未知系统和自适应滤波器的振幅特性")
    pl.xlabel(u"圆频率")
    pl.ylabel(u"振幅(dB)")
    pl.show()
```

如果延迟器的延时D不够的话，会由于因果律使得自适应滤波器无法收敛。因此这里我们采用的D的长度为h0的长度的2倍。下图显示h0, h和它们的卷积。我们看到h0和h的卷积正好是一个脉冲，其延时为正好等于D(128)。



未知系统和自适应滤波器的级联(卷积)近似为标准延迟

下图显示未知系统的频率响应(蓝色)和自适应滤波器的频率响应(绿色)，我们看到二者正好相反，也就是说自适应滤波器均衡了未知系统对信号的影响。



未知系统和自适应滤波器的频率响应正好相反

卷积逆运算

虽然卷积运算最终能归结为简单的加法和乘法运算，然而卷积的逆运算就不是很容易计算了。我们知道两个线性系统h1和h2的级联h3可以用它们的脉冲响应的卷积计算求得，而所谓卷积的逆运算可以想象为已知h3和h1，求一个h2使它和h1级联之后正好等于h3。

根据卷积的计算公式可知，如果h1的长度为100，h3的长度为199，那么h2的长度则为100，因为h2的每个系数都是未知的，于是就有100个未知数，而这100个未知数需要满足199个线性方程：h3中的每个系数都有一个方程与之对应。由于方程数大于未知数的个数，显然对于任意的h1和h3并不能保证有一个h2使得它和h1的卷积正好等于h3。

既然不能精确求解，那么卷积的逆运算就变成了一个误差最小化的优化问题。用自适应滤波器计算卷积的逆运算和计算信号均衡类似，将白色噪声x输入到h1中得到信号u，将x输入到h3中得到信号d，然后使用u作为参照信号，d作为目标信号进行NLMS计算，最终收敛后的自适应滤波器的系数就是h2。

下面的程序模拟这一过程：

```
# -*- coding: utf-8 -*-
import numpy as np
import pylab as pl
from nlms_numpy import nlms
import scipy.signal as signal

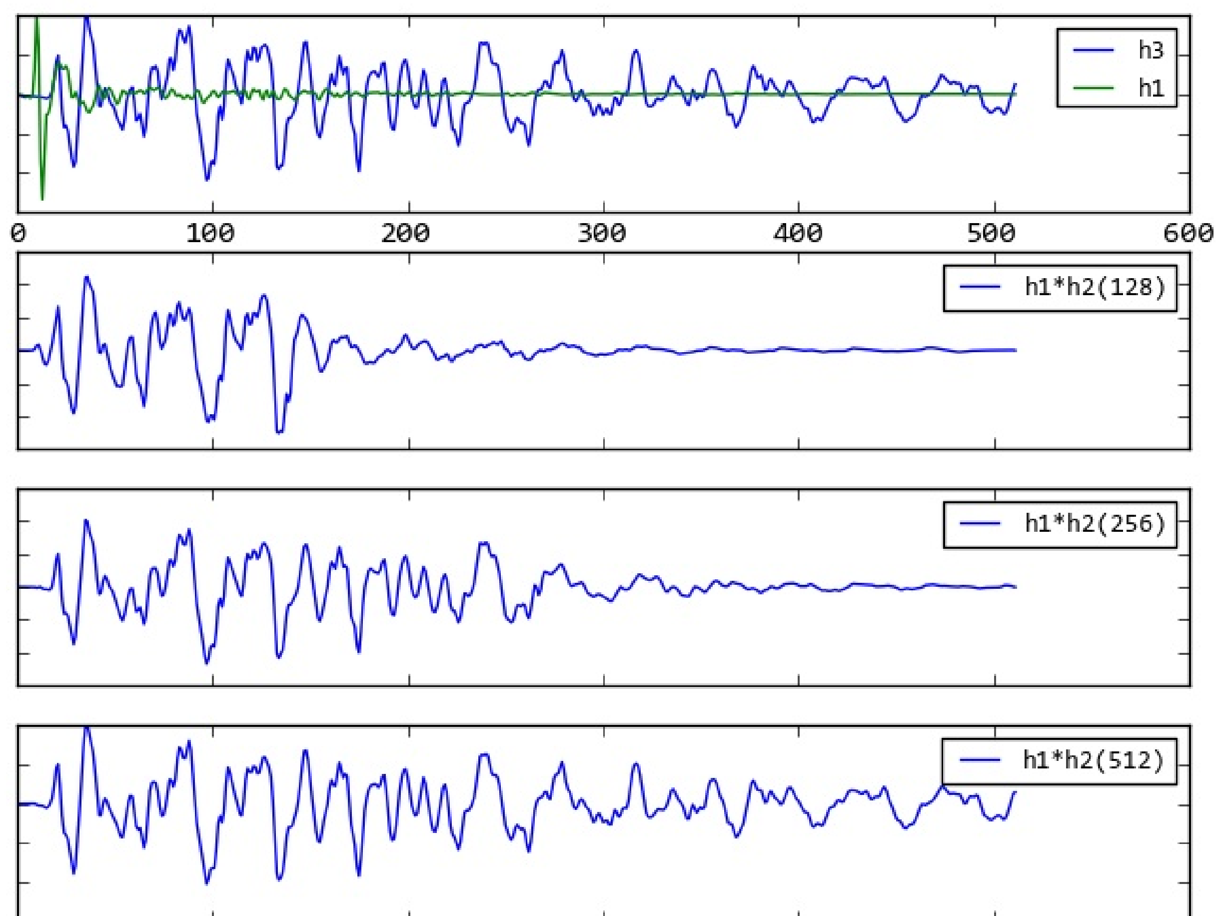
def inv_convolve(h1, h3, length):
    x = np.random.standard_normal(10000)
    u = signal.lfilter(h1, 1, x)
    d = signal.lfilter(h3, 1, x)
    h = np.zeros(length, np.float64)
    nlms(u, d, h, 0.1)
    return h

h1 = np.fromfile("h1.txt", sep="\n")
h1 /= np.max(h1)
h3 = np.fromfile("h3.txt", sep="\n")
h3 /= np.max(h3)

pl.rc('legend', fontsize=10)
pl.subplot(411)
pl.plot(h3, label="h3")
pl.plot(h1, label="h1")
pl.legend()
pl.gca().set_yticklabels([])
for idx, length in enumerate([128, 256, 512]):
    pl.subplot(412+idx)
    h2 = inv_convolve(h1, h3, length)
    pl.plot(np.convolve(h1, h2)[:len(h3)], label="h1*h2(%s)" % leng
    pl.legend()
    pl.gca().set_yticklabels([])
    pl.gca().set_xticklabels([])

pl.show()
```

下面是程序的计算结果：



卷积逆运算演示

程序中的h1和h3从文本文件中读取而得，它们是ANC(能动噪声控制)系统中实际测量的脉冲响应。如果能找到一个h2满足卷积条件的话，就能够有效的进行噪声控制。

程序计算出h2的长度分别为128, 256, 512时的结果，可以看出h2越长结果越精确。

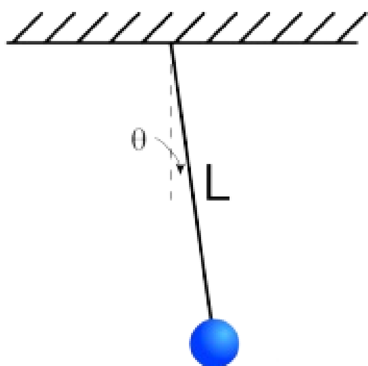
DLL函数的编写

ctypes的python接口

单摆和双摆模拟

单摆模拟

由一根不可伸长、质量不计的绳子，上端固定，下端系一质点，这样的装置叫做单摆。



单摆装置示意图

根据牛顿力学定律，我们可以列出如下微分方程：

$$\frac{d^2\theta}{dt^2} + \frac{g}{\ell}\sin\theta = 0$$

其中 θ 为单摆的摆角，ℓ 为单摆的长度， g为重力加速度。

此微分方程的符号解无法直接求出，因此只能调用odeint对其求数值解。

odeint函数的调用参数如下：

```
odeint(func, y0, t, ...)
```

其中func是一个Python的函数对象，用来计算微分方程组中每个未知函数的导数，y0为微分方程组中每个未知函数的初始值，t为需要进行数值求解的时间点。它返回的是一个二维数组result，其第0轴的长度为t的长度，第1轴的长度为变量的个数，因此 result[:, i] 为第i个未知函数的解。

计算微分的func函数的调用参数为： func(y, t)，其中y是一个数组，为每个未知函数在t时刻的值，而func的返回值也是数组，它为每个未知函数在t时刻的导数。

odeint要求每个微分方程只包含一阶导数，因此我们需要对前面的微分方程做如下的变形：

$$\frac{d\theta(t)}{dt} = v(t)$$

$$\frac{dv(t)}{dt} = -\frac{g}{\ell}\sin\theta(t)$$

下面是利用odeint计算单摆轨迹的程序：

```
# -*- coding: utf-8 -*-

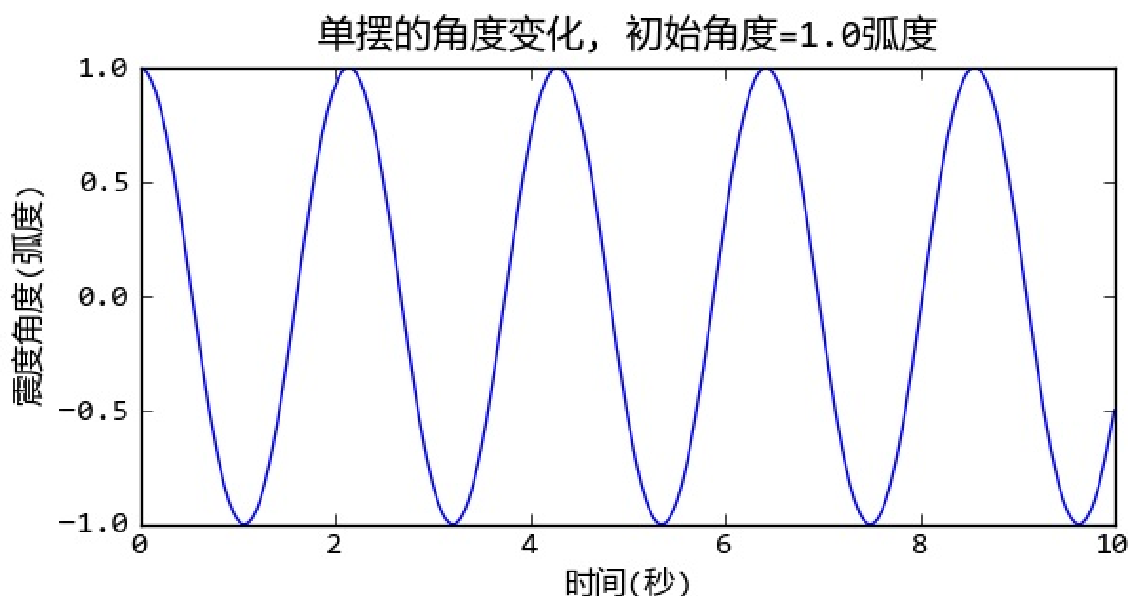
from math import sin
import numpy as np
from scipy.integrate import odeint

g = 9.8

def pendulum_equations(w, t, l):
    th, v = w
    dth = v
    dv  = - g/l * sin(th)
    return dth, dv

if __name__ == "__main__":
    import pylab as pl
    t = np.arange(0, 10, 0.01)
    track = odeint(pendulum_equations, (1.0, 0), t, args=(1.0,))
    pl.plot(t, track[:, 0])
    pl.title(u"单摆的角度变化, 初始角度=1.0弧度")
    pl.xlabel(u"时间(秒)")
    pl.ylabel(u"震度角度(弧度)")
    pl.show()
```

odeint函数还有一个关键字参数args，其值为一个组元，这些值都会作为额外的参数传递给func函数。程序使用这种方式将单摆的长度传递给pendulum_equations函数。



初始角度为1弧度的单摆摆动角度和时间的关系

计算摆动周期

高中物理课介绍过当最大摆动角度很小时，单摆的摆动周期可以使用如下公式计算：

$$T_0 = 2\pi\sqrt{\frac{\ell}{g}}$$

这是因为当 $\theta << 1$ 时， $\sin\theta \approx \theta$， 这样微分方程就变成了：

$$\frac{d^2\theta}{dt^2} + \frac{g}{\ell}\theta = 0$$

此微分方程的解是一个简谐震动方程，很容易计算其摆动周期。但是当初始摆角增大时，上述的近似处理会带来无法忽视的误差。下面让我们来看看如何用数值计算的方法求出单摆在任意初始摆角时的摆动周期。

要计算摆动周期只需要计算从最大摆角到0摆角所需的时间，摆动周期是此时间的4倍。为了计算出这个时间值，首先需要定义一个函数pendulum_th计算任意时刻的摆角：

```
def pendulum_th(t, l, th0):
    track = odeint(pendulum_equations, (th0, 0), [0, t], args=(l,)
    return track[-1, 0]
```

pendulum_th函数计算长度为l初始角度为th0的单摆在时刻t的摆角。此函数仍然使用odeint进行微分方程组求解，只是我们只需要计算时刻t的摆角，因此传递给odeint的时间序列为[0, t]。 odeint内部会对时间进行细分，保证最终的解是正确的。

接下来只需要找到第一个时pendulum_th的结果为0的时间即可。这相当于对pendulum_th函数求解，可以使用 scipy.optimize.fsolve 函数对这种非线性方程进行求解。

```
def pendulum_period(l, th0):
    t0 = 2*np.pi*sqrt( l/g ) / 4
    t = fsolve( pendulum_th, t0, args = (l, th0) )
    return t*4
```

和odeint一样，我们通过fsolve的args关键字参数将额外的参数传递给pendulum_th函数。fsolve求解时需要一个初始值尽量接近真实的解，用小角度单摆的周期的1/4作为这个初始值是一个很不错的选择。下面利用pendulum_period函数计算出初始摆动角度从0到90度的摆动周期：

```
ths = np.arange(0, np.pi/2.0, 0.01)
periods = [pendulum_period(1, th) for th in ths]
```

为了验证fsolve求解摆动周期的正确性，我从维基百科中找到摆动周期的精确解：

$$T = 4\sqrt{\frac{\ell}{g}}K\left(\sin\frac{\theta_0}{2}\right)$$

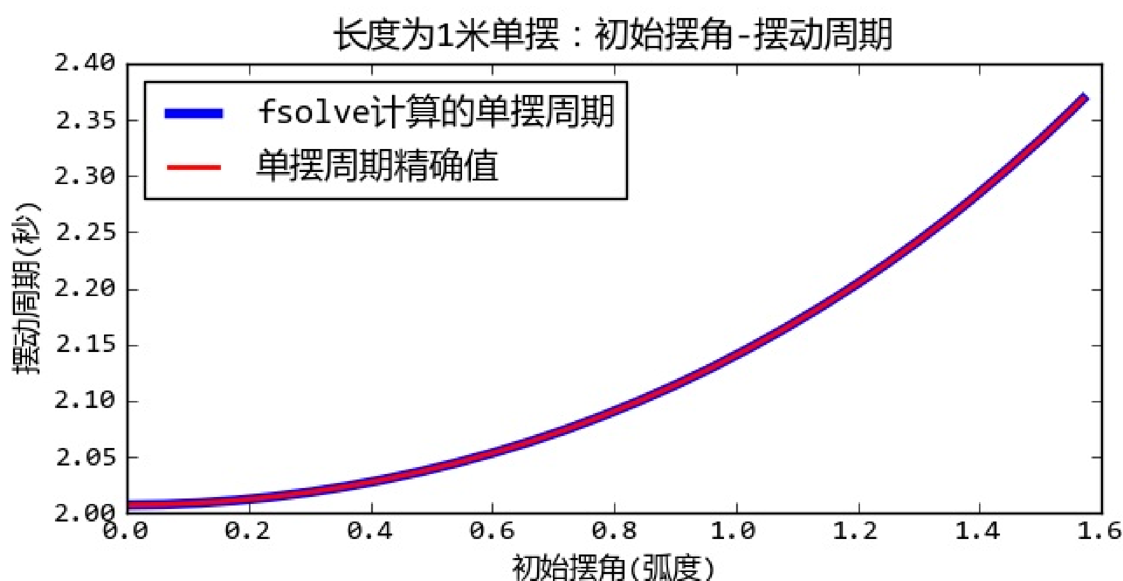
其中的函数K为第一类完全椭圆积分函数，其定义如下：

$$K(k) = \int_0^{\pi/2} \frac{\mathrm{d}\theta}{\sqrt{1-k^2\sin^2\theta}}$$

我们可以用 scipy.special.ellipk 来计算此函数的值：

```
periods2 = 4*sqrt(1.0/g)*ellipk(np.sin(ths/2)**2)
```

下图比较两种计算方法，我们看到其结果是完全一致的：

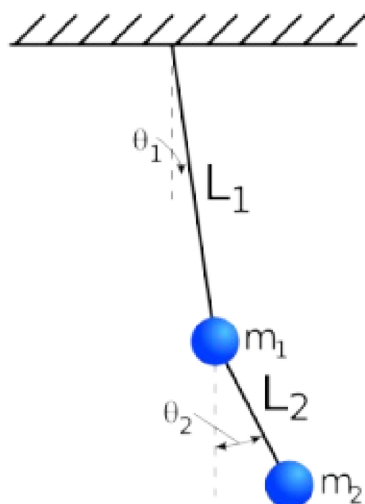


单摆的摆动周期和初始角度的关系

完整的程序请参见： 单摆摆动周期的计算

双摆模拟

接下来让我们来看看如何对双摆系统进行模拟。双摆系统的如下图所示，



双摆装置示意图

两根长度为L1和L2的无质量的细棒的顶端有质量分别为m1和m2的两个球，初始角度为 θ_1 和 θ_2， 要求计算从此初始状态释放之后的两个球的运动轨迹。

公式推导

本节首先介绍如何利用拉格朗日力学获得双摆系统的微分方程组。

拉格朗日力学(摘自维基百科)

拉格朗日力学是分析力学中的一种。於 1788 年由拉格朗日所创立，拉格朗日力学是对经典力学的一种的新的理论表述。

经典力学最初的表述形式由牛顿建立，它着重於分析位移，速度，加速度，力等矢量间的关系，又称为矢量力学。拉格朗日引入了广义坐标的概念，又运用达朗贝尔原理，求得与牛顿第二定律等价的拉格朗日方程。不仅如此，拉格朗日方程具有更普遍的意义，适用范围更广泛。还有，选取恰当的广义坐标，可以大大地简化拉格朗日方程的求解过程。

假设杆L1连接的球体的坐标为x1和y1，杆L2连接的球体的坐标为x2和y2，那么x1,y1,x2,y2和两个角度之间有如下关系：

$$x_1 = L_1 \sin(\theta_1)$$

$$y_1 = -L_1 \cos(\theta_1)$$

$$x_2 = L_1 \sin(\theta_1) + L_2 \sin(\theta_2)$$

$$y_2 = -L_1 \cos(\theta_1) - L_2 \cos(\theta_2)$$

根据拉格朗日量的公式：

$$\mathcal{L} = T - V$$

其中T为系统的动能，V为系统的势能，可以得到如下公式：

$$\mathcal{L} = \frac{m_1}{2}(\dot{x}_1^2 + \dot{y}_1^2) + \frac{m_2}{2}(\dot{x}_2^2 + \dot{y}_2^2) - m_1 g y_1 - m_2 g y_2$$

其中正号的项目为两个小球的动能，符号的项目为两个小球的势能。

将前面的坐标和角度之间的关系公式带入之后整理可得：

![\mathcal{L} = \frac{m_1+m_2}{2} L_1^2 {\dot \theta_1}^2 + \frac{m_2}{2} L_2^2 {\dot \theta_2}^2 + m_2 L_1 L_2 {\dot \theta_1} {\dot \theta_2} \cos(\theta_1 - \theta_2) +

(m_1 + m_2) g L_1 \cos(\theta_1) + m_2 g L_2 \cos(\theta_2)](img/ddb810d6eed716ca3b15c0a3e9c94185accee394.png)

对于变量 θ_1 的拉格朗日方程：

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{\theta}_1} - \frac{\partial \mathcal{L}}{\partial \theta_1} = 0$$

得到：

$$L_1[(m_1+m_2)L_1\ddot{\theta}_1 + m_2 L_2 \cos(\theta_1-\theta_2)\ddot{\theta}_2 + m_2 L_2 \sin(\theta_1-\theta_2)\dot{\theta}_2^2 + (m_1+m_2)g\sin(\theta_1)] = 0$$

对于变量 θ_2 的拉格朗日方程：

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{\theta}_2} - \frac{\partial \mathcal{L}}{\partial \theta_2} = 0$$

得到：

$$m_2 L_2[L_2\ddot{\theta}_2 + L_1\cos(\theta_1-\theta_2)\ddot{\theta}_1 - L_1\sin(\theta_1-\theta_2)\dot{\theta}_1^2 + g\sin(\theta_2)] = 0$$

这一计算过程可以用sympy进行推导：

```
# -*- coding: utf-8 -*-
from sympy import *
from sympy import Derivative as D

var("x1 x2 y1 y2 l1 l2 m1 m2 th1 th2 dth1 dth2 ddth1 ddth2 t g tmp'

sublist = [
(D(th1(t), t, t), ddth1),
(D(th1(t), t), dth1),
(D(th2(t), t, t), ddth2),
(D(th2(t),t), dth2),
(th1(t), th1),
(th2(t), th2)
]

x1 = l1*sin(th1(t))
y1 = -l1*cos(th1(t))
x2 = l1*sin(th1(t)) + l2*sin(th2(t))
y2 = -l1*cos(th1(t)) - l2*cos(th2(t))

vx1 = diff(x1, t)
vx2 = diff(x2, t)
vy1 = diff(y1, t)
vy2 = diff(y2, t)

# 拉格朗日量
L = m1/2*(vx1**2 + vy1**2) + m2/2*(vx2**2 + vy2**2) - m1*g*y1 - m2

# 拉格朗日方程
def lagrange_equation(L, v):
    a = L.subs(D(v(t), t), tmp).diff(tmp).subs(tmp, D(v(t), t))
    b = L.subs(D(v(t), t), tmp).subs(v(t), v).diff(v).subs(v, v(t)
    c = a.diff(t) - b
    c = c.subs(sublist)
    c = trigsimp(simplify(c))
    c = collect(c, [th1,th2,dth1,dth2,ddth1,ddth2])
    return c

eq1 = lagrange_equation(L, th1)
eq2 = lagrange_equation(L, th2)
```

执行此程序之后，eq1对应于 θ_1 的拉格朗日方程， eq2对应于 θ_2 的方程。

由于sympy只能对符号变量求导数，即只能计算 D(L, t), 而不能计算D(f, v(t))。 因此在求偏导数之前，将偏导数变量置换为一个tmp变量，然后对tmp变量求导数，例如下面的程序行对D(v(t), t)求偏导数，即计算 $\partial\mathcal{L}/\partial\dot{v}$

```
L.subs(D(v(t), t), tmp).diff(tmp).subs(tmp, D(v(t), t))
```

而在计算 $\partial\mathcal{L}/\partial v$ 时，需要将v(t)替换为v之后再进行微分计算。由于将v(t)替换为v的同时，会将 D(v(t), t) 中的也进行替换，这是我们不希望的结果，因此先将 D(v(t), t) 替换为tmp，微分计算完毕之后再替换回去：

```
L.subs(D(v(t), t), tmp).subs(v(t), v).diff(v).subs(v, v(t)).subs(tr
```

最后得到的eq1, eq2的值为：

```
>>> eq1
ddth1*(m1*l1**2 + m2*l1**2) +
ddth2*(l1*l2*m2*cos(th1)*cos(th2) + l1*l2*m2*sin(th1)*sin(th2)) +
dth2**2*(l1*l2*m2*cos(th2)*sin(th1) - l1*l2*m2*cos(th1)*sin(th2)) -
g*l1*m1*sin(th1) + g*l1*m2*sin(th1)
>>> eq2
ddth1*(l1*l2*m2*cos(th1)*cos(th2) + l1*l2*m2*sin(th1)*sin(th2)) +
dth1**2*(l1*l2*m2*cos(th1)*sin(th2) - l1*l2*m2*cos(th2)*sin(th1)) -
g*l2*m2*sin(th2) + ddth2*m2*l2**2
```

结果看上去挺复杂，其实只要运用如下的三角公式就和前面的结果一致了：

![\sin \left(x+y\right)=\sin x \cos y + \cos x \sin y

\cos \left(x+y\right)=\cos x \cos y - \sin x \sin y

\sin \left(x-y\right)=\sin x \cos y - \cos x \sin y

\cos \left(x-y\right)=\cos x \cos y + \sin x \sin y](img/5149d766733feeccc71ce88a0b4d99749b762842.png)

微分方程的数值解

接下来要做的事情就是对如下的微分方程求数值解：

$$(m_1+m_2)L_1\ddot{\theta}_1 + m_2L_2\cos(\theta_1-\theta_2)\ddot{\theta}_2 + m_2L_2\sin(\theta_1-\theta_2)\dot{\theta}_2^2 + (m_1+m_2)g\sin(\theta_1) = 0$$

$$L_2\ddot{\theta}_2 + L_1\cos(\theta_1-\theta_2)\ddot{\theta}_1 - L_1\sin(\theta_1-\theta_2)\dot{\theta}_1^2 + g\sin(\theta_2) = 0$$

由于方程中包含二阶导数，因此无法直接使用odeint函数进行数值求解，我们很容易将其改写为4个一阶微分方程组，4个未知变量为： $\theta_1, \theta_2, v_1, v_2$， 其中 v_1, v_2 为两个杆转动的角速度。

$$\dot{\theta}_1 = v_1$$

$$\dot{\theta}_2 = v_2$$

$$(m_1+m_2)L_1\dot{v}_1 + m_2L_2\cos(\theta_1-\theta_2)\dot{v}_2 + m_2L_2\sin(\theta_1-\theta_2)\dot{\theta}_2^2 + (m_1+m_2)g\sin(\theta_1) = 0$$

$$L_2\dot{v}_2 + L_1\cos(\theta_1-\theta_2)\dot{v}_1 - L_1\sin(\theta_1-\theta_2)\dot{\theta}_1^2 + g\sin(\theta_2) = 0$$

下面的程序利用 scipy.integrate.odeint 对此微分方程组进行数值求解：

```
# -*- coding: utf-8 -*-

from math import sin,cos
import numpy as np
from scipy.integrate import odeint

g = 9.8

class DoublePendulum(object):
    def __init__(self, m1, m2, l1, l2):
        self.m1, self.m2, self.l1, self.l2 = m1, m2, l1, l2
        self.init_status = np.array([0.0,0.0,0.0,0.0])

    def equations(self, w, t):
        """
 微分方程公式
 """
        m1, m2, l1, l2 = self.m1, self.m2, self.l1, self.l2
        th1, th2, v1, v2 = w
        dth1 = v1
        dth2 = v2

        #eq of th1
        a = l1*l1*(m1+m2)  # dv1 parameter
        b = l1*m2*l2*cos(th1-th2) # dv2 paramter
        c = l1*(m2*l2*sin(th1-th2)*dth2*dth2 + (m1+m2)*g*sin(th1))

        #eq of th2
        d = m2*l2*l1*cos(th1-th2) # dv1 parameter
        e = m2*l2*l2 # dv2 parameter
        f = m2*l2*(-l1*sin(th1-th2)*dth1*dth1 + g*sin(th2))

        dv1, dv2 = np.linalg.solve([[a,b],[d,e]], [-c,-f])

        return np.array([dth1, dth2, dv1, dv2])

def double_pendulum_odeint(pendulum, ts, te, tstep):
    """
 对双摆系统的微分方程组进行数值求解，返回两个小球的X-Y坐标
 """
    t = np.arange(ts, te, tstep)
    track = odeint(pendulum.equations, pendulum.init_status, t)
    th1_array, th2_array = track[:,0], track[:, 1]
    l1, l2 = pendulum.l1, pendulum.l2
    x1 = l1*np.sin(th1_array)
    y1 = -l1*np.cos(th1_array)
    x2 = x1 + l2*np.sin(th2_array)
    y2 = y1 - l2*np.cos(th2_array)
    pendulum.init_status = track[-1,:].copy() #将最后的状态赋给pendul
    return [x1, y1, x2, y2]
```

```
if __name__ == "__main__":
    import matplotlib.pyplot as pl
    pendulum = DoublePendulum(1.0, 2.0, 1.0, 2.0)
    th1, th2 = 1.0, 2.0
    pendulum.init_status[:2] = th1, th2
    x1, y1, x2, y2 = double_pendulum_odeint(pendulum, 0, 30, 0.02)
    pl.plot(x1,y1, label = u"上球")
    pl.plot(x2,y2, label = u"下球")
    pl.title(u"双摆系统的轨迹, 初始角度=%s,%s" % (th1, th2))
    pl.legend()
    pl.axis("equal")
    pl.show()
```

程序中的 DoublePendulum.equations 函数计算各个未知函数的导数，其输入参数w数组中的变量依次为：

- th1: 上球角度
- th2: 下球角度
- v1: 上球角速度
- v2: 下球角速度

返回值为每个变量的导数：

- dth1: 上球角速度
- dth2: 下球角速度
- dv1: 上球角加速度
- dv2: 下球角加速度

其中dth1和dth2很容易计算，它们直接等于传入的角速度变量：

```
dth1 = v1
dth2 = v2
```

为了计算dv1和dv2，需要将微分方程组进行变形为如下格式：

![\dot v_1 = ...

\dot v_2 = ...](img/47827c6cd286f5e2e5d906f8e99385366e0db189.png)

如果我们希望让程序做这个事情的话，可以计算出 dv1 和 dv2 的系数，然后调用 linalg.solve 求解线型方程组：

```
#eq of th1
a = l1*l1*(m1+m2)  # dv1 parameter
b = l1*m2*l2*cos(th1-th2) # dv2 paramter
c = l1*(m2*l2*sin(th1-th2)*dth2*dth2 + (m1+m2)*g*sin(th1))

#eq of th2
d = m2*l2*l1*cos(th1-th2) # dv1 parameter
e = m2*l2*l2 # dv2 parameter
f = m2*l2*(-l1*sin(th1-th2)*dth1*dth1 + g*sin(th2))

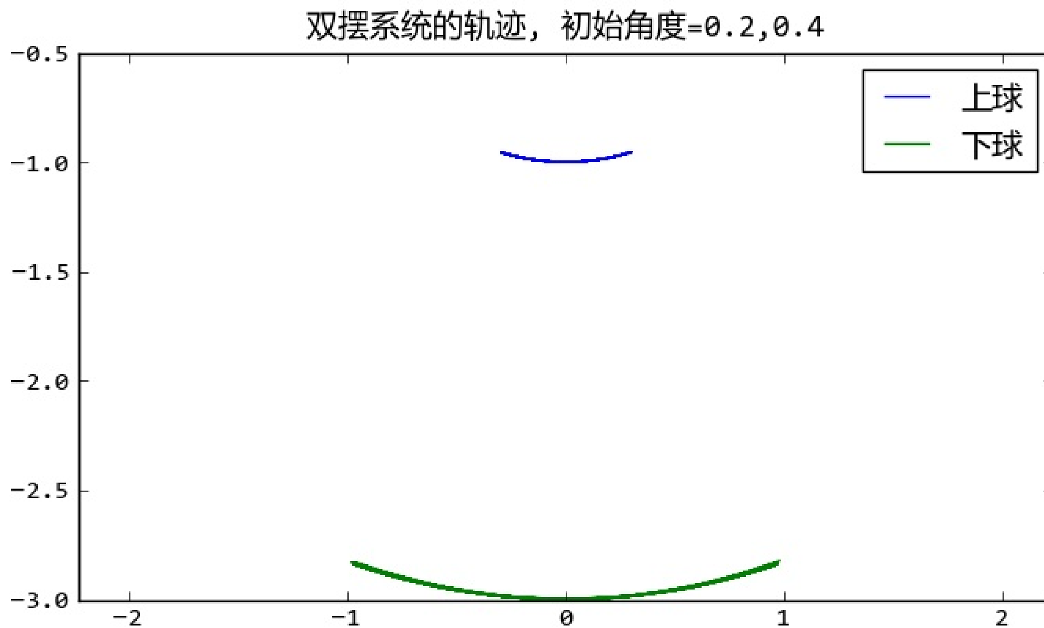
dv1, dv2 = np.linalg.solve([[a,b],[d,e]], [-c,-f])
```

上面的程序相当于将原始的微分方程组变换为

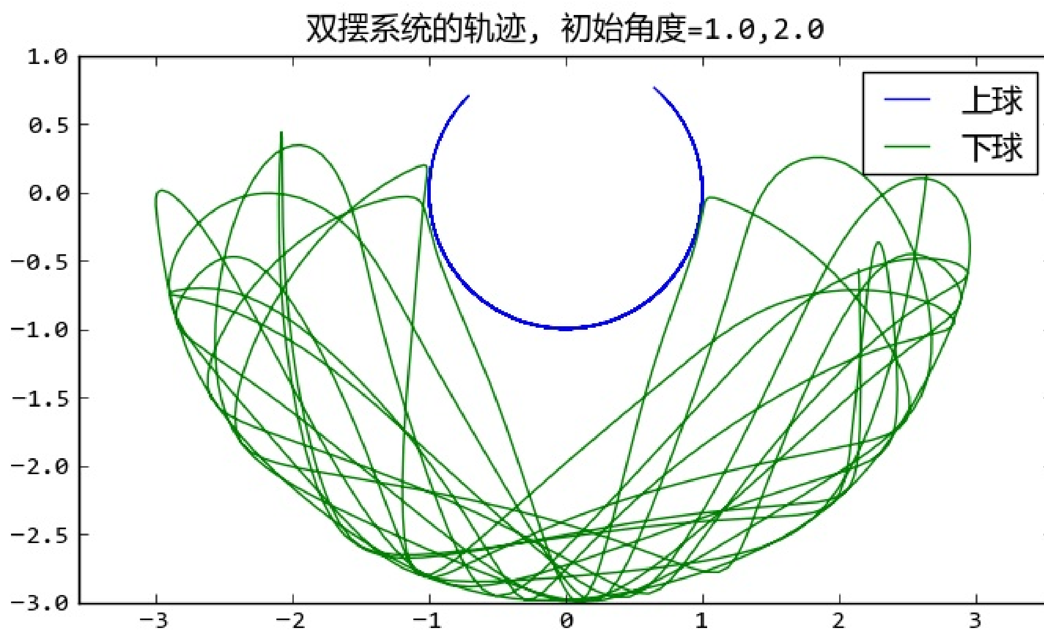
![a \dot v_1 + b \dot v_2 + c = 0

d \dot v_1 + e \dot v_2 + f = 0](img/0adfbc12bbe1ff9d4511bc29a532781e32e7609e.png)

程序绘制的小球运动轨迹如下：



初始角度微小时的双摆的摆动轨迹



大初始角度时双摆的摆动轨迹呈现混沌现象

可以看出当初始角度很大的时候，摆动出现混沌现象。

动画显示

计算出小球的轨迹之后我们很容易将结果可视化，制作成动画效果。制作动画可以有多种选择：

- visual库可以制作3D动画
- pygame制作快速的2D动画
- tkinter或者wxpython直接在界面上绘制动画

这里介绍如何使用matplotlib制作动画。整个动画绘制程序如下：

```
# -*- coding: utf-8 -*-
import matplotlib
matplotlib.use('WXAgg') # do this before importing pylab
import matplotlib.pyplot as pl
from double_pendulum_odeint import double_pendulum_odeint, DoublePe

fig = pl.figure(figsize=(4,4))
line1, = pl.plot([0,0], [0,0], "-o")
line2, = pl.plot([0,0], [0,0], "-o")
pl.axis("equal")
pl.xlim(-4,4)
pl.ylim(-4,2)

pendulum = DoublePendulum(1.0, 2.0, 1.0, 2.0)
pendulum.init_status[:] = 1.0, 2.0, 0, 0

x1, y1, x2, y2 = [],[],[],[]
idx = 0

def update_line(event):
    global x1, x2, y1, y2, idx
    if idx == len(x1):
        x1, y1, x2, y2 = double_pendulum_odeint(pendulum, 0, 1, 0.0
        idx = 0
    line1.set_xdata([0, x1[idx]])
    line1.set_ydata([0, y1[idx]])
    line2.set_xdata([x1[idx], x2[idx]])
    line2.set_ydata([y1[idx], y2[idx]])
    fig.canvas.draw()
    idx += 1

import wx
id = wx.NewId()
actor = fig.canvas.manager.frame
timer = wx.Timer(actor, id=id)
timer.Start(1)
wx.EVT_TIMER(actor, id, update_line)
pl.show()
```

程序中强制使用WXAgg进行后台绘制：

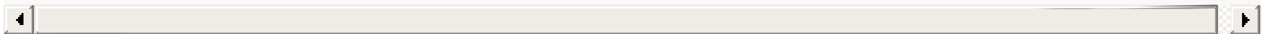
```
matplotlib.use('WXAgg')
```

然后启动wx库中的时间事件调用update_line函数重新设置两条直线的端点位置：

```
import wx
id = wx.NewId()
actor = fig.canvas.manager.frame
timer = wx.Timer(actor, id=id)
timer.Start(1)
wx.EVT_TIMER(actor, id, update_line)
```

在update_line函数中，每次轨迹数组播放完毕之后，就调用：

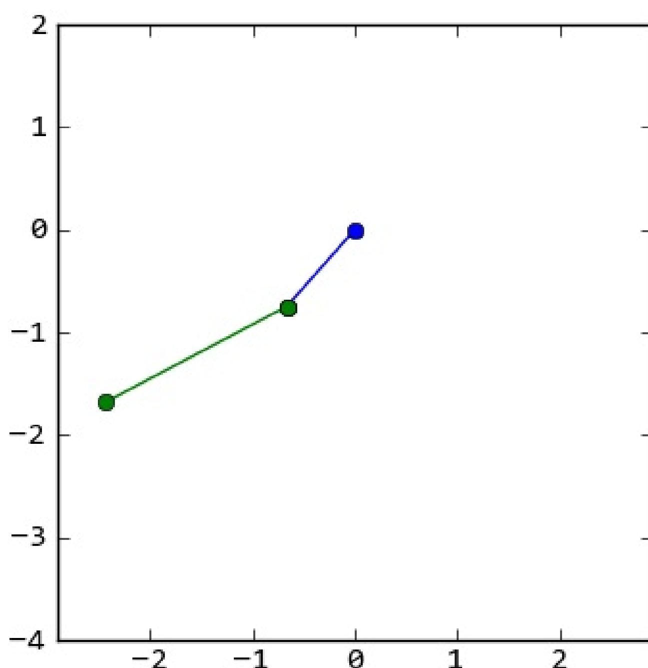
```
if idx == len(x1):
    x1, y1, x2, y2 = double_pendulum_odeint(pendulum, 0, 1, 0.05)
    idx = 0
```



重新生成下一秒钟的轨迹。由于在 double_pendulum_odeint 函数中会将odeint计算的最终的状态赋给 pendulum.init_status， 因此连续调用 double_pendulum_odeint 函数可以生成连续的运动轨迹

```
def double_pendulum_odeint(pendulum, ts, te, tstep):
    ...
    track = odeint(pendulum.equations, pendulum.init_status, t)
    ...
    pendulum.init_status = track[-1,:].copy()
    return [x1, y1, x2, y2]
```

程序的动画效果如下图所示：



双摆的摆动动画效果截图

分形与混沌

自然界的很多事物，例如树木、云彩、山脉、闪电、雪花以及海岸线等等都呈现出传统的几何学不能描述的形状。这些形状都有如下的特性：

- 有着十分精细的不规则的结构
- 整体与局部相似，例如一根树杈的形状和一棵树很像

分形几何学就是用来研究这样一类的几何形状的科学，借助计算机的高速计算和图像显示，使得我们可以更加深入地直观地观察分形几何。在本章中，让我们用Python绘制一些经典的分形图案。

Mandelbrot集合

Mandelbrot(曼德布洛特)集合是在复平面上组成分形的点的集合。

Mandelbrot集合的定义(摘自维基百科)

Mandelbrot集合可以用下面的复二次多项式定义：

$f_c(z) = z^2 + c$

其中c是一个复参数。对于每一个c，从z=0开始对函数 $f_c(z)$ 进行迭代。

序列 $(0, f_c(0), f_c(f_c(0)), f_c(f_c(f_c(0))), \ldots)$ 的值或者延伸到无限大，或者只停留在有限半径的圆盘内。

Mandelbrot集合就是使以上序列不发散的所有c点的集合。

从数学上来讲，Mandelbrot集合是一个复数的集合。一个给定的复数c或者属于Mandelbrot集合，或者不是。

用程序绘制Mandelbrot集合时不能进行无限次迭代，最简单的方法是使用逃逸时间(迭代次数)进行绘制，具体算法如下：

- 判断每次调用函数 $f_c(z)$ 得到的结果是否在半径R之内，即复数的模小于R
- 记录下模大于R时的迭代次数
- 迭代最多进行N次
- 不同的迭代次数的点使用不同的颜色绘制

下面是完整的绘制Mandelbrot集合的程序：

```
# -*- coding: utf-8 -*-

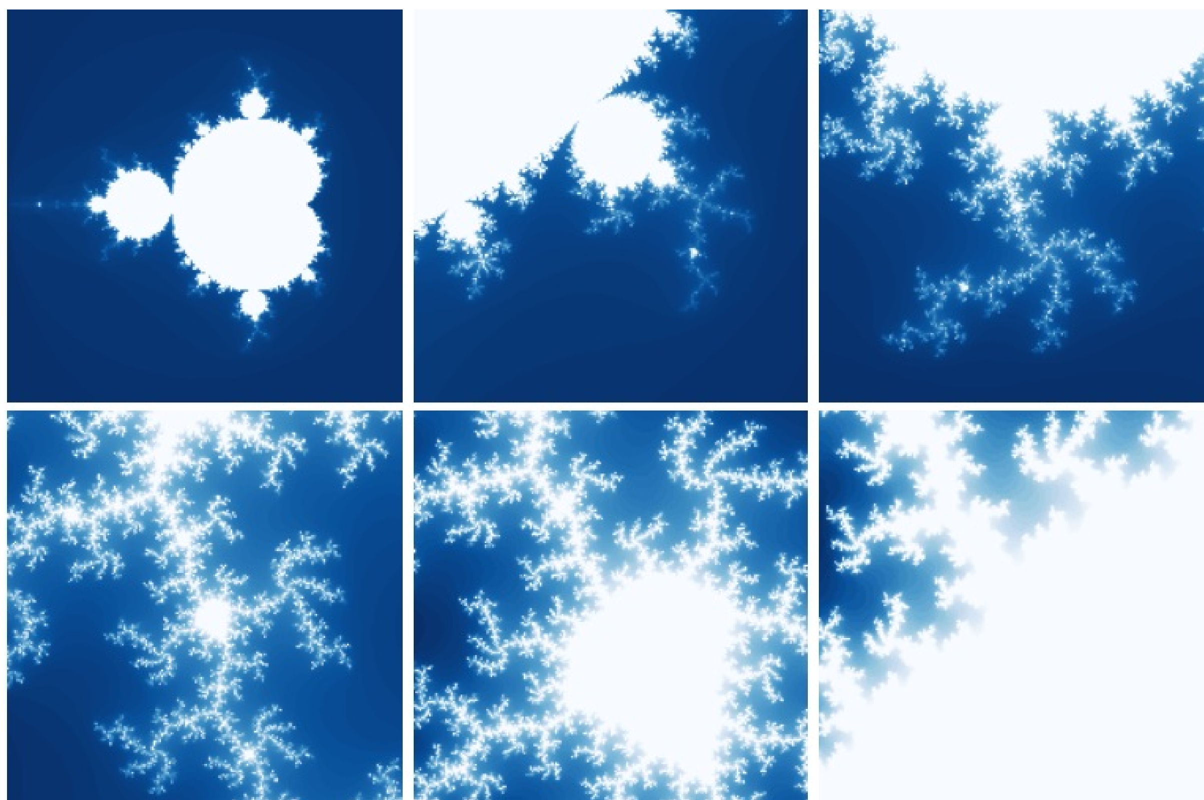
import numpy as np
import pylab as pl
import time
from matplotlib import cm

def iter_point(c):
    z = c
    for i in xrange(1, 100): # 最多迭代100次
        if abs(z)>2: break # 半径大于2则认为逃逸
        z = z*z+c
    return i # 返回迭代次数

def draw_mandelbrot(cx, cy, d):
    """
  绘制点(cx, cy)附近正负d的范围的Mandelbrot
  """
    x0, x1, y0, y1 = cx-d, cx+d, cy-d, cy+d
    y, x = np.ogrid[y0:y1:200j, x0:x1:200j]
    c = x + y*1j
    start = time.clock()
    mandelbrot = np.frompyfunc(iter_point,1,1)(c).astype(np.float)
    print "time=",time.clock() - start
    pl.imshow(mandelbrot, cmap=cm.Blues_r, extent=[x0,x1,y0,y1])
    pl.gca().set_axis_off()

x,y = 0.27322626, 0.595153338

pl.subplot(231)
draw_mandelbrot(-0.5,0,1.5)
for i in range(2,7):
    pl.subplot(230+i)
    draw_mandelbrot(x, y, 0.2**(i-1))
pl.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0)
pl.show()
```



Mandelbrot集合，以5倍的倍率放大点(0.273, 0.595)附近

程序中的iter_point函数计算点c的逃逸时间，逃逸半径R为2.0，最大迭代次数为100。draw_mandelbrot函数绘制以点(cx, cy)为中心，边长为2*d的正方形区域内的Mandelbrot集合。

下面3行计算指定范围内的迭代公式的参数c，c是一个元素为复数的二维数组，大小为200*200，注意np.ogrid不是函数：

```
x0, x1, y0, y1 = cx-d, cx+d, cy-d, cy+d
y, x = np.ogrid[y0:y1:200j, x0:x1:200j]
c = x + y*1j
```

下面一行程序通过调用np.frompyfunc将iter_point转换为NumPy的ufunc函数，这样它可以自动对c中的每个元素调用iter_point函数，由于结果的数组元素类型为object，还需要调用astype方法将其元素类型转换为浮点类型：

```
mandelbrot = np.frompyfunc(iter_point,1,1)(c).astype(np.float)
```

最后调用matplotlib的imshow函数将结果数组绘制成图，通过cmap关键字参数指定图的值和颜色的映射表：

```
pl.imshow(mandelbrot, cmap=cm.Blues_r, extent=[x0,x1,y0,y1])
```

使用Python绘制Mandelbrot集合最大的问题就是运算速度太慢，下面是上面每幅图的计算时间：

```
time= 0.88162629608
time= 1.53712748408
time= 1.71502160191
time= 1.8691174437
time= 3.03812691278
```

因为计算每个点的逃逸时间均不相同，因此每幅图的计算时间也不相同。

计算速度慢的最大的原因是因为iter_point函数的运算速度慢，如果将此函数用C语言重写的话将能显著地提高计算速度，下面使用scipy.weave库将C++重写的iter_point函数转换为Python能调用的函数：

```
import scipy.weave as weave

def weave_iter_point(c):
    code = """
 std::complex<double> z;
 int i;
 z = c;
 for(i=1;i<100;i++)
 {
 if(std::abs(z) > 2) break;
 z = z*z+c;
 }
 return_val=i;
 """

    f = weave.inline(code, ["c"], compiler="gcc")
    return f
```

下面是使用weave_iter_point函数计算Mandelbrot集合的时间：

```
time= 0.285266982256
time= 0.271430028118
time= 0.293769180161
time= 0.308515188383
time= 0.411168179196
```

通过NumPy的数组运算也可以提高计算速度，前面的计算都是先对复数平面上的每个点进行循环，然后再循环迭代计算每个点的逃逸时间。如果要用NumPy的数组运算加速计算的话，可以将这两个循环的顺序颠倒过来，下面的程序演示这一算法：

```
# -*- coding: utf-8 -*-
```

```
import numpy as np
import pylab as pl
import time
from matplotlib import cm

def draw_mandelbrot(cx, cy, d, N=200):
    """
 绘制点(cx, cy)附近正负d的范围的Mandelbrot
 """
    global mandelbrot

    x0, x1, y0, y1 = cx-d, cx+d, cy-d, cy+d
    y, x = np.ogrid[y0:y1:N*1j, x0:x1:N*1j]
    c = x + y*1j

    # 创建X,Y轴的坐标数组
    ix, iy = np.mgrid[0:N,0:N]

    # 创建保存mandelbrot图的二维数组，缺省值为最大迭代次数
    mandelbrot = np.ones(c.shape, dtype=np.int)*100

    # 将数组都变成一维的
    ix.shape = -1
    iy.shape = -1
    c.shape = -1
    z = c.copy() # 从c开始迭代，因此开始的迭代次数为1

    start = time.clock()

    for i in xrange(1,100):
        # 进行一次迭代
        z *= z
        z += c
        # 找到所有结果逃逸了的点
        tmp = np.abs(z) > 2.0
        # 将这些逃逸点的迭代次数赋值给mandelbrot图
        mandelbrot[ix[tmp], iy[tmp]] = i

        # 找到所有没有逃逸的点
        np.logical_not(tmp, tmp)
        # 更新ix, iy, c, z只包含没有逃逸的点
        ix,iy,c,z = ix[tmp], iy[tmp], c[tmp],z[tmp]
        if len(z) == 0: break

    print "time=",time.clock() - start
    pl.imshow(mandelbrot, cmap=cm.Blues_r, extent=[x0,x1,y0,y1])
    pl.gca().set_axis_off()

x,y = 0.27322626, 0.595153338

pl.subplot(231)
draw_mandelbrot(-0.5,0,1.5)
for i in range(2,7):
```

```
    pl.subplot(230+i)
    draw_mandelbrot(x, y, 0.2**(i-1))
pl.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0)
pl.show()
```

为了减少计算次数，程序中每次迭代之后，都将已经逃逸的点剔除出去，这样就需要保存每个点的下标，程序中用ix和iy这两个数组来保存没有逃逸的点的下标，因为有额外的数组保存下标，因此数组z和c不需要是二维的。函数迭代部分的程序如下：

```
# 进行一次迭代
z *= z
z += c
```

使用 *=, += 这样的运算符能够让NumPy不分配额外的空间直接在数组z上进行运算。

下面的程序计算出逃逸点，tmp是逃逸点在z中的下标，由于z和ix和iy等数组始终是同时更新的，因此ix[tmp], iy[tmp]就是逃逸点在图像中的下标：

```
# 找到所有结果逃逸了的点
tmp = np.abs(z) > 2.0
# 将这些逃逸点的迭代次数赋值给mandelbrot图
mandelbrot[ix[tmp], iy[tmp]] = i
```

最后通过对tmp中的每个元素取逻辑反，更新所有没有逃逸的点的对应的ix, iy, c, z：

```
# 找到所有没有逃逸的点
np.logical_not(tmp, tmp)
# 更新ix, iy, c, z只包含没有逃逸的点
ix,iy,c,z = ix[tmp], iy[tmp], c[tmp], z[tmp]
```

此程序的计算时间如下：

```
time= 0.186070576008
time= 0.327006365334
time= 0.372756034636
time= 0.410074464771
time= 0.681048289658
time= 0.878626752841
```

连续的逃逸时间

修改逃逸半径R和最大迭代次数N，可以绘制出不同效果的Mandelbrot集合图案。但是前面所述的方法计算出的逃逸时间是大于逃逸半径时的迭代次数，因此所输出的图像最多只有N种不同的颜色值，有很强的梯度感。为了在不同的梯度之间进行渐变处理，使用下面的公式进行逃逸时间计算：

$$n - \log_2 \log_2 |z_n|$$

z_n 是迭代n次之后的结果，通过在逃逸时间的计算中引入迭代结果的模值，结果将不再是整数，而是平滑渐变的。

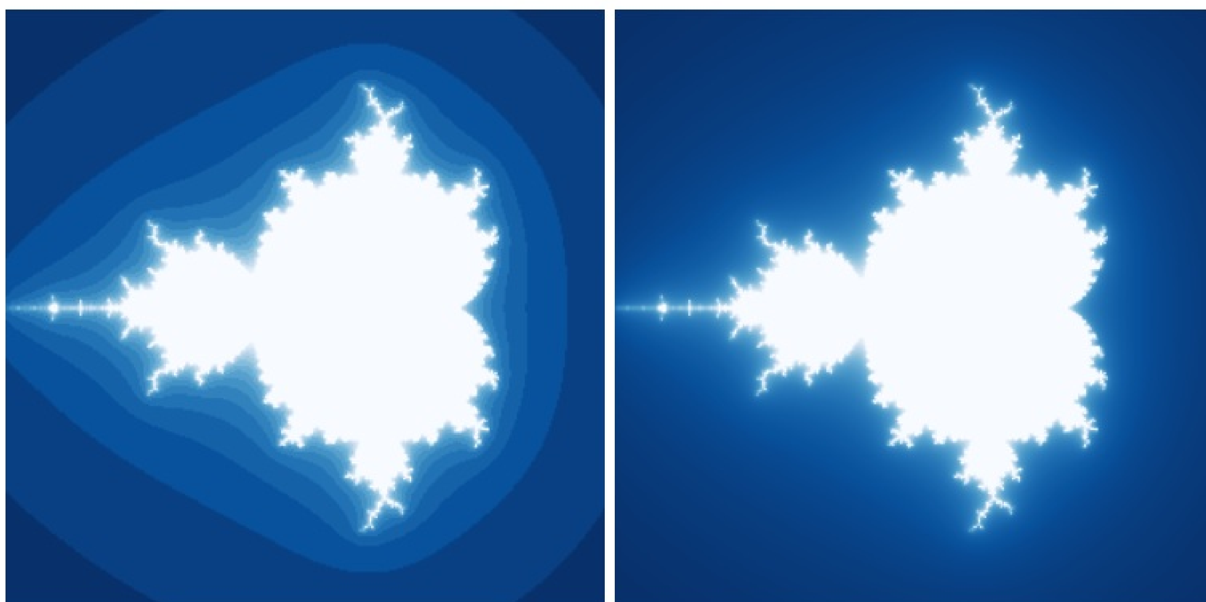
下面是计算此逃逸时间的程序：

```
def smooth_iter_point(c):
    z = c
    for i in xrange(1, iter_num):
        if abs(z)>escape_radius: break
        z = z*z+c
    absz = abs(z)
    if absz > 2.0:
        mu = i - log(log(abs(z),2),2)
    else:
        mu = i
    return mu # 返回正规化的迭代次数
```

如果你的逃逸半径设置得很小，例如2.0，那么有可能结果不够平滑，这时可以在迭代循环之后添加几次迭代保证z能够足够逃逸，例如：

```
z = z*z+c
z = z*z+c
i += 2
```

下图是逃逸半径为10，最大迭代次数为20时，绘制的结果：



逃逸半径=10，最大迭代次数=20的平滑处理后的Mandelbrot集合

逃逸时间公式是如何得出的？

请参考： http://linas.org/art-gallery/escape/ray.html

完整的程序请参考 绘制*Mandelbrot*集合

迭代函数系统(IFS)

迭代函数系统是一种用来创建分形图案的算法，它所创建的分形图永远是绝对自相似的。下面我们直接通过绘制一种蕨类植物的叶子来说明迭代函数系统的算法：

有下面4个线性函数将二维平面上的坐标进行线性映射变换：

```
1.
    x(n+1) = 0
    y(n+1) = 0.16 * y(n)

2.
    x(n+1) = 0.2 * x(n) - 0.26 * y(n)
    y(n+1) = 0.23 * x(n) + 0.22 * y(n) + 1.6

3.
    x(n+1) = -0.15 * x(n) + 0.28 * y(n)
    y(n+1) = 0.26 * x(n) + 0.24 * y(n) + 0.44

4.
    x(n+1) = 0.85 * x(n) + 0.04 * y(n)
    y(n+1) = -0.04 * x(n) + 0.85 * y(n) + 1.6
```

所谓迭代函数是指将函数的输出再次当作输入进行迭代计算，因此上面公式都是通过坐标 x(n),y(n) 计算变换后的坐标 x(n+1),y(n+1)。现在的问题是有4个迭代函数，迭代时选择哪个函数进行计算呢？我们为每个函数指定一个概率值，它们依次为1%, 7%, 7%和85%。选择迭代函数时使用通过每个函数的概率随机选择一个函数进行迭代。上面的例子中，第四个函数被选择迭代的概率最高。

最后我们从坐标原点(0,0)开始迭代，将每次迭代所得到的坐标绘制成图，就得到了叶子的分形图案。下面的程序演示这一计算过程：

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as pl
import time

# 蕨类植物叶子的迭代函数和其概率值
eq1 = np.array([[0,0,0],[0,0.16,0]])
p1 = 0.01
```

```
eq2 = np.array([[0.2,-0.26,0],[0.23,0.22,1.6]])
p2 = 0.07

eq3 = np.array([[-0.15, 0.28, 0],[0.26,0.24,0.44]])
p3 = 0.07

eq4 = np.array([[0.85, 0.04, 0],[-0.04, 0.85, 1.6]])
p4 = 0.85

def ifs(p, eq, init, n):
    """
    进行函数迭代
    p: 每个函数的选择概率列表
    eq: 迭代函数列表
    init: 迭代初始点
    n: 迭代次数

    返回值： 每次迭代所得的X坐标数组， Y坐标数组， 计算所用的函数下标
    """

    # 迭代向量的初始化
    pos = np.ones(3, dtype=np.float)
    pos[:2] = init

    # 通过函数概率，计算函数的选择序列
    p = np.add.accumulate(p)
    rands = np.random.rand(n)
    select = np.ones(n, dtype=np.int)*(n-1)
    for i, x in enumerate(p[::-1]):
        select[rands<x] = len(p)-i-1

    # 结果的初始化
    result = np.zeros((n,2), dtype=np.float)
    c = np.zeros(n, dtype=np.float)

    for i in xrange(n):
        eqidx = select[i] # 所选的函数下标
        tmp = np.dot(eq[eqidx], pos) # 进行迭代
        pos[:2] = tmp # 更新迭代向量

        # 保存结果
        result[i] = tmp
        c[i] = eqidx

    return result[:,0], result[:, 1], c

start = time.clock()
x, y, c = ifs([p1,p2,p3,p4],[eq1,eq2,eq3,eq4], [0,0], 100000)
print time.clock() - start
pl.figure(figsize=(6,6))
pl.subplot(121)
pl.scatter(x, y, s=1, c="g", marker="s", linewidths=0)
pl.axis("equal")
```

```
pl.axis("off")
pl.subplot(122)
pl.scatter(x, y, s=1,c = c, marker="s", linewidths=0)
pl.axis("equal")
pl.axis("off")
pl.subplots_adjust(left=0,right=1,bottom=0,top=1,wspace=0,hspace=0)
pl.gcf().patch.set_facecolor("white")
pl.show()
```

程序中的ifs函数是进行函数迭代的主函数，我们希望通过矩阵乘法计算函数(numpy.dot)的输出，因此需要将乘法向量扩充为三维的：

```
pos = np.ones(3, dtype=np.float)
pos[:2] = init
```

这样每次和迭代函数系数进行矩阵乘积运算的向量就变成了： x(n), y(n), 1.0 。

为了减少计算时间，我们不在迭代循环中计算随机数选择迭代方程，而是事先通过每个函数的概率，计算出函数选择数组select，注意这里使用accumulate函数先将概率累加，然后产生一组0到1之间的随机数，通过判断随机数所在的概率区间选择不同的方程下标：

```
p = np.add.accumulate(p)
rands = np.random.rand(n)
select = np.ones(n, dtype=np.int)*(n-1)
for i, x in enumerate(p[::-1]):
    select[rands<x] = len(p)-i-1
```

最后我们通过调用scatter绘图函数将所得到的坐标进行散列图绘制：

```
pl.scatter(x, y, s=1, c="g", marker="s", linewidths=0)
```

其中每个关键字参数的含义如下：

- **s** : 散列点的大小，因为我们要绘制10万点，因此大小选择为1
- **c** : 点的颜色，这里选择绿色
- **marker** : 点的形状，"s"表示正方形，方形的绘制是最快的
- **linewidths** : 点的边框宽度，0表示没有边框

此外，关键字参数c还可以传入一个数组，作为每个点的颜色值，我们将计算坐标的函数下标传入，这样可以直观地看出哪个点是哪个函数迭代产生的：

```
pl.scatter(x, y, s=1,c = c, marker="s", linewidths=0)
```

下图是程序的输出：



函数迭代系统所绘制的蕨类植物的叶子

观察右图的4种颜色的部分可以发现概率为1%的函数1所计算的是叶杆部分(深蓝色)，概率为7%的两个函数计算的是左右两片子叶，而概率为85%的函数计算的是整个叶子的迭代：即最下面的三种颜色的点通过此函数的迭代产生上面的所有的深红色的点。

我们可以看出整个叶子呈现出完美的自相似特性，任意取其中的一个子叶，将其旋转放大之后都和整个叶子相同。

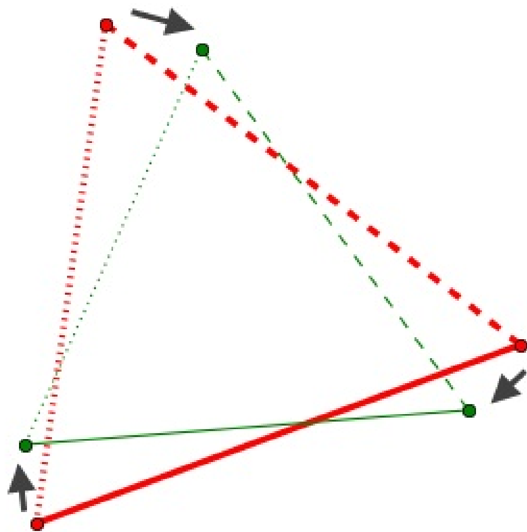
2D仿射变换

上面所介绍的4个变换方程的一般形式如下：

```
x(n+1) = A * x(n) + B * y(n) + C
y(n+1) = D * x(n) + E * y(n) + F
```

这种变换被称为2D仿射变换，它是从2D坐标到其他2D坐标的线性映射，保留直线性和平行性。即原来是直线上的坐标，变换之后仍然成一条直线，原来是平行的直线，变换之后仍然是平行的。这种变换我们可以看作是一系列平移、缩放、翻转和旋转变换构成的。

为了直观地显示仿射变换，我们可以使用平面上的两个三角形来表示。因为仿射变换公式中有6个未知数：A, B, C, D, E, F，而每两个点之间的变换决定两个方程，因此一共需要3组点来决定六个变换方程，正好是两个三角形，如下图所示：

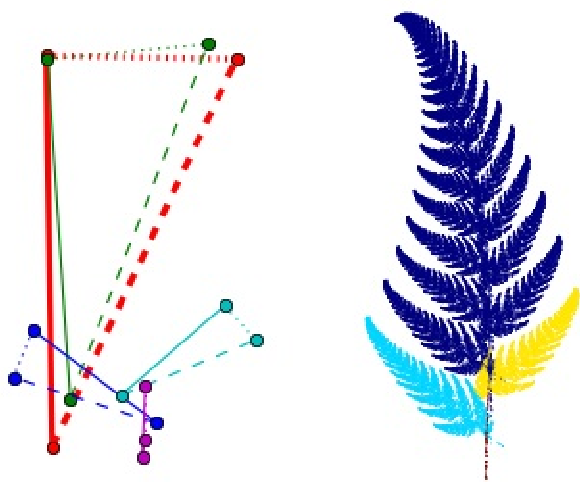


两个三角形决定一个2D仿射变换的六个参数

从红色三角形的每个顶点变换到绿色三角形的对应顶点，正好能够决定仿射变换中的六个参数。这样我们可是使用N+1个三角形，决定N个仿射变换，其中的每一个变换的参数都是由第0个三角形和其它的三角形决定的。这第0个三角形我们称之为基础三角形，其余的三角形称之为变换三角形。

为了绘制迭代函数系统的图像，我们还需要给每个仿射变换方程指定一个迭代概率的参数。此参数也可以使用三角形直观地表达出来：迭代概率和变换三角形的面积成正比。即迭代概率为变换三角形的面积除以所有变换三角形的面积之和。

如下图所示，前面介绍的蕨类植物的分形图案的迭代方程可以由5个三角形决定，可以很直观地看出紫色的小三角形决定了叶子的茎；而两个蓝色的三角形决定了左右两片子叶；绿色的三角形将茎和两片子叶往上复制，形成整片叶子。



5个三角形的仿射方程绘制蕨类植物的叶子

迭代函数系统设计器

按照上节所介绍的三角形法，我们可以设计一个迭代函数系统的设计工具，如下图所示：



具体的程序请参照 迭代函数系统的分形 ，这里简单地介绍一下程序的几个核心组成部分：

首先通过两个三角形求解仿射方程的系数相当于求六元线性方程组的解，solve_eq函数完成这一工作，它先计算出线性方程组的矩阵a和b, 然后调用NumPy的linalg.solve对线性方程组 a*X = b 求解：

```
def solve_eq(triangle1, triangle2):
    """
 解方程，从triangle1变换到triangle2的变换系数
 triangle1,2是二维数组：
 x0,y0
 x1,y1
 x2,y2
 """
    x0,y0 = triangle1[0]
    x1,y1 = triangle1[1]
    x2,y2 = triangle1[2]

    a = np.zeros((6,6), dtype=np.float)
    b = triangle2.reshape(-1)
    a[0, 0:3] = x0,y0,1
    a[1, 3:6] = x0,y0,1
    a[2, 0:3] = x1,y1,1
    a[3, 3:6] = x1,y1,1
    a[4, 0:3] = x2,y2,1
    a[5, 3:6] = x2,y2,1

    c = np.linalg.solve(a, b)
    c.shape = (2,3)
    return c
```

triangle_area函数计算三角形的面积，它使用NumPy的cross函数计算三角形的两个边的向量的叉积：

```
def triangle_area(triangle):
    """
 计算三角形的面积
 """
    A = triangle[0]
    B = triangle[1]
    C = triangle[2]
    AB = A-B
    AC = A-C
    return np.abs(np.cross(AB,AC))/2.0
```

整个程序的界面使用TraitsUI库生成，将matplotlib的Figure控件通过MPLFigureEditor和*MPLFigureEditor*类嵌入到*TraitsUI*生成的界面中，请参考：*[*设计自己的Trait编辑器_](traitsui_manual_custom_editor.html)

IFSDesigner._figure_default创建Figure对象，并且添加两个并排的子图ax和ax2，ax用于三角形编辑，而ax2用于分形图案显示。

```
def _figure_default(self):
    """
 figure属性的缺省值，直接创建一个Figure对象
 """
    figure = Figure()
    self.ax = figure.add_subplot(121)
    self.ax2 = figure.add_subplot(122)
    self.ax2.set_axis_off()
    self.ax.set_axis_off()
    figure.subplots_adjust(left=0,right=1,bottom=0,top=1,wspace=0,h
    figure.patch.set_facecolor("w")
    return figure
```

IFSTriangles类完成三角形的编辑工作，其中通过如下的语句绑定Figure控件的canvas的鼠标事件

```
canvas = ax.figure.canvas
# 绑定canvas的鼠标事件
canvas.mpl_connect('button_press_event', self.button_press_callback
canvas.mpl_connect('button_release_event', self.button_release_call
canvas.mpl_connect('motion_notify_event', self.motion_notify_callba
```

由于canvas只有在真正显示Figure时才会创建，因此不能在创建Figure控件时创建IFSTriangles对象，而需要在界面生成之后，显示之前创建它。这里我们通过给IFSDesigner类的view属性指定其handler为IFSHandler对象，重载Handler的init方法，此方法在界面生成之后，显示之前被调用：

```
class IFSHandler(Handler):
    """
 在界面显示之前需要初始化的内容
 """
    def init(self, info):
        info.object.init_gui_component()
        return True
```

然后IFSDesigner类的init_gui_component方法完成实际和canvas相关的初始工作：

```
def init_gui_component(self):
    self.ifs_triangle = IFSTriangles(self.ax)
    self.figure.canvas.draw()
    thread.start_new_thread( self.ifs_calculate, ())
    ...
```

由于通过函数迭代计算分形图案比较费时，因此在另外一个线程中执行ifs_calculate方法进行运算，每计算ITER_COUNT个点，就调用ax2.scatter将产生的点添加进ax2中，由于随着ax2中的点数增加，界面重绘将越来越慢，因此在draw_points函数中限制最多只调用ITER_TIMES次scatter函数。因为在别的线程中不能更新界面，因此通过调用wx.CallAfter在管理GUI的线程中调用draw_points进行界面刷新。:

```
def ifs_calculate(self):
    """
 在别的线程中计算
 """
    def draw_points(x, y, c):
        if len(self.ax2.collections) < ITER_TIMES:
            try:
                self.ax2.scatter(x, y, s=1, c=c, marker="s", linewi
                self.ax2.set_axis_off()
                self.ax2.axis("equal")
                self.figure.canvas.draw()
            except:
                pass

    def clear_points():
        self.ax2.clear()

    while 1:
        try:
            if self.exit == True:
                break
            if self.clear == True:
                self.clear = False
                self.initpos = [0, 0]
                x, y, c = ifs( self.ifs_triangle.get_areas(),
                    self.ifs_triangle.get_eqs(), self.initpos, 100)
                self.initpos = [x[-1], y[-1]]
                self.ax2.clear()

            x, y, c = ifs( self.ifs_triangle.get_areas(),
                self.ifs_triangle.get_eqs(), self.initpos, ITER_COU
            if np.max(np.abs(x)) < 1000000 and np.max(np.abs(y)) <
                self.initpos = [x[-1], y[-1]]
                wx.CallAfter( draw_points, x, y, c )
            time.sleep(0.05)
        except:
            pass
```

用户修改三角形之后，需要重新迭代，并绘制分形图案，三角形的改变通过IFSTriangles.version 属性通知给IFSDesigner，在IFSTriangles中，三角形改变之后，将运行：

```
self.version += 1
```

在IFSDesigner中监听version属性的变化：

```
@on_trait_change("ifs_triangle.version")
def on_ifs_version_changed(self):
    """
 当三角形更新时，重新绘制所有的迭代点
 """
    self.clear = True
```

当IFSDesigner.clear为True时，真正进行迭代运算的ifs_calculate方法就知道需要重新计算了。

L-System分形

前面所绘制的分形图案都是都是使用数学函数的迭代产生，而L-System分形则是采用符号的递归迭代产生。首先如下定义几个有含义的符号：

- **F** : 向前走固定单位
- **+** : 正方向旋转固定单位
- **-** : 负方向旋转固定单位

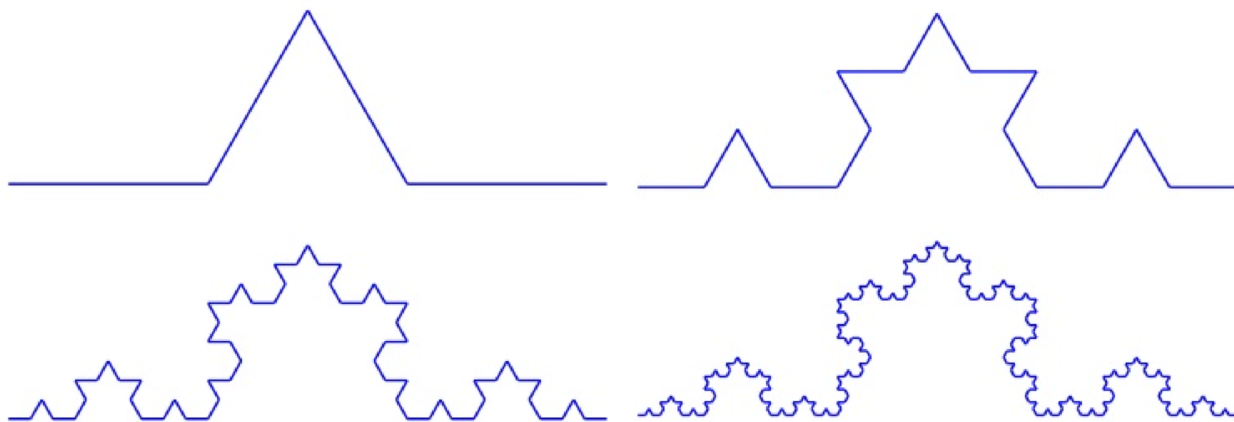
使用这三个符号我们很容易描述下图中由4条线段构成的图案：

```
F+F--F+F
```

如果将此符号串中的所有F都替换为F+F--F+F，就能得到如下的新字符串：

```
F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F
```

如此替换迭代下去，并根据字串进行绘图(符号+和-分别正负旋转60度)，可得到如下的分形图案：



使用F+F--F+F迭代的分形图案

除了 F, +, - 之外我们再定义如下几个符号：

- **f** : 向前走固定单位，为了定义不同的迭代公式
- **[** : 将当前的位置入堆栈
- **]** : 从堆栈中读取坐标，修改当前位置
- **S** : 初始迭代符号

所有的符号(包括上面未定义的)都可以用来定义迭代，通过引入两个方括号符号，使得我们能够描述分岔的图案。

例如下面的符号迭代能够绘制出一棵植物：

```
S -> X
X -> F-[[X]+X]+F[+FX]-X
F -> FF
```

我们用一个字典定义所有的迭代公式和其它的一些绘图信息：

```
{
    "X":"F-[[X]+X]+F[+FX]-X", "F":"FF", "S":"X",
    "direct":-45,
    "angle":25,
    "iter":6,
    "title":"Plant"
}
```

其中：

- **direct** : 是绘图的初始角度，通过指定不同的值可以旋转整个图案
- **angle** : 定义符号+,-旋转时的角度，不同的值能产生完全不同的图案
- **iter** : 迭代次数

下面的程序将上述字典转换为需要绘制的线段坐标：

```
class L_System(object):
    def __init__(self, rule):
        info = rule['S']
        for i in range(rule['iter']):
            ninfo = []
            for c in info:
                if c in rule:
                    ninfo.append(rule[c])
                else:
                    ninfo.append(c)
            info = "".join(ninfo)
        self.rule = rule
        self.info = info

    def get_lines(self):
        d = self.rule['direct']
        a = self.rule['angle']
        p = (0.0, 0.0)
        l = 1.0
        lines = []
        stack = []
        for c in self.info:
            if c in "Ff":
                r = d * pi / 180
                t = p[0] + l*cos(r), p[1] + l*sin(r)
                lines.append(((p[0], p[1]), (t[0], t[1])))
                p = t
            elif c == "+":
                d += a
            elif c == "-":
                d -= a
            elif c == "[":
                stack.append((p,d))
            elif c == "]":
                p, d = stack[-1]
                del stack[-1]
        return lines
```

我们使用matplotlib的LineCollection绘制所有的直线：

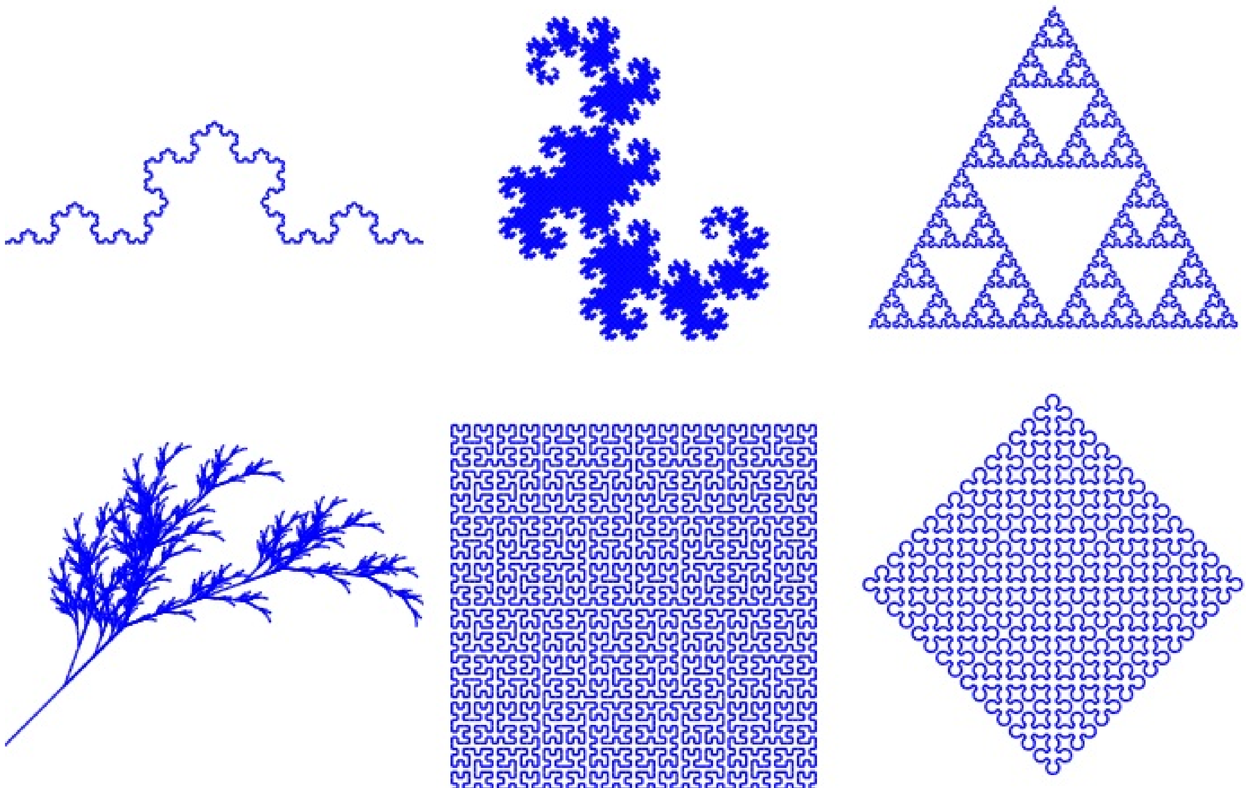
```
import matplotlib.pyplot as pl
from matplotlib import collections

# rule = {...}  此处省略rule的定义

lines = L_System(rule).get_lines()

fig = pl.figure()
ax = fig.add_subplot(111)
linecollections = collections.LineCollection(lines)
ax.add_collection(linecollections, autolim=True)
pl.show()
```

下面是几种L-System的分形图案，绘制此图的完整程序请参照 绘制*L-System*的分形图 。



几种L-System的迭代图案

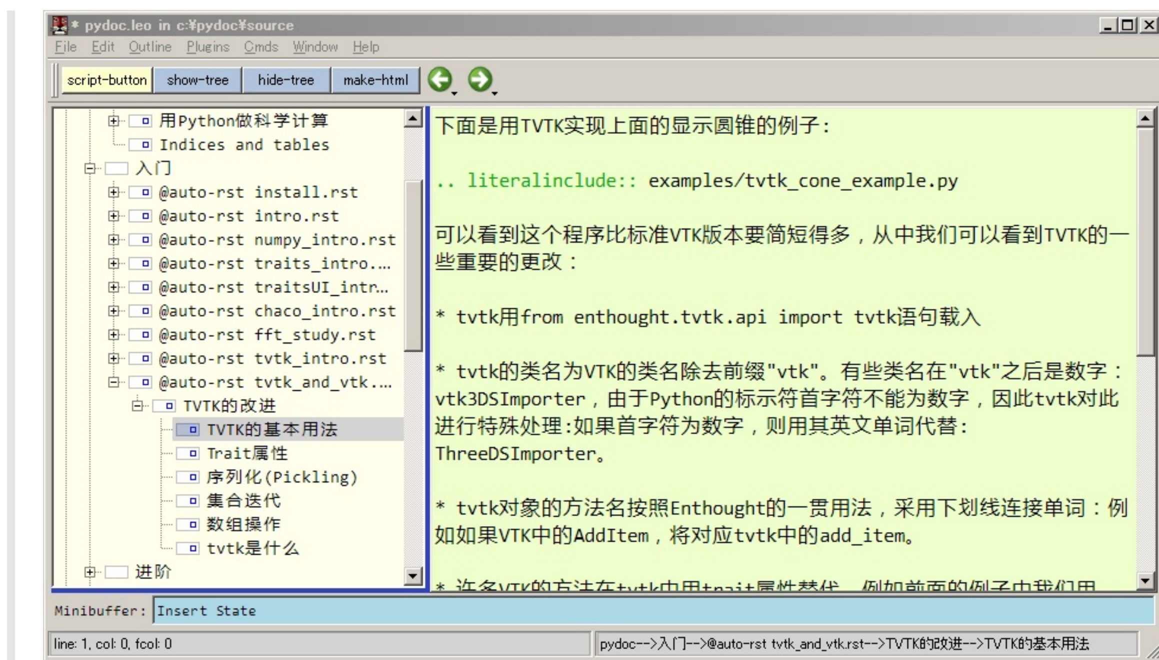
关于本书的编写

为了编写此书，我评价了许多写书的软件，最终决定使用Sphinx和reStructuredText作为写书的工具。随着章节的逐渐增加，我越来越觉得当初的选择是正确的。

本书的编写工具

本书采用reStructuredText(rst) 格式的文本编写，然后用Sphinx将reStructuredText文件自动转换为html格式的文件。采用Leo管理和组织所有的文档。用proTeXt将latex格式的数学公式转换为PNG图片。

- **reStructuredText**：一种结构化文本格式，它提供了对写书所需的各种元素的支持。例如章节、链接、格式、图片以及语法高亮等等。
- **Sphinx**：将一系列reStructuredText文本转换成各种不同的输出格式，并自动制作交叉引用（cross-references)、索引等。也就是说，如果某目录中有一系列的reStructuredText格式的文档，Sphinx可以制作一份组织得非常完美的HTML文件。
- **Leo**：以树状结构管理文本、代码的编辑器，可以用它来进行数据组织和项目管理。我使用它管理构成本书的所有rst文档、python程序以及图片和笔记。下面是使用Leo编写本书时的一个例子：

  编写本书所使用的Leo编辑器的界面

- **PicPick**, **Greenshot**：界面截图工具。

问题与解决方案

在使用上述工具编写本书时，为了达到完美的效果，我对这些工具做了一些配置和修改的工作。

代码中的注释

Sphinx使用Pygments进行代码高亮的处理，在Pygments的缺省样式中，代码注释部分是采用斜体字表示的，斜体的汉字看起来十分别扭，因此需要将缺省样式的斜体改为正体。在conf.py文件中有如下配置：

```
# The name of the Pygments (syntax highlighting) style to use.
pygments_style = 'sphinx'
```

它指定pygments使用sphinx样式对代码进行高亮处理，我没有弄明白如何添加自己定义的样式，因此直接手工修改定义此样式的文件：

```
%Python安装目录%\Lib\site-packages\sphinx\highlighting.py
```

将其中的Comment的样式改为noitalic：

```
...
styles.update({
    Generic.Output: '#333',
    Comment: 'noitalic #408090',
    Number: '#208050',
})
...
```

修改**Sphinx**的主题

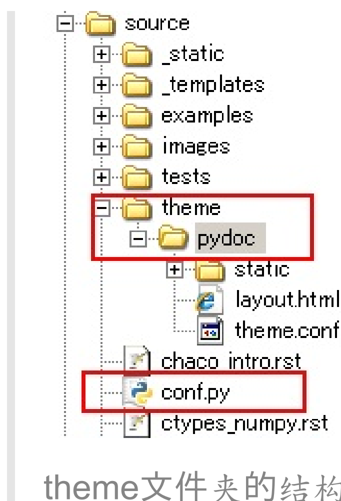
为了给文档添加评论功能，必须添加一部分javascript代码，因此需要修改Shpinx的主题。

- 首先编辑conf.py文件中如下的两个配置：

```
# The theme to use for HTML and HTML Help pages.  Major themes
# Sphinx are currently 'default' and 'sphinxdoc'.
html_theme = 'pydoc'

# Add any paths that contain custom themes here, relative to th
html_theme_path = ["./theme"]
```

- 然后在conf.py文件所在的目录下创建一个子目录theme，将sphinx安装目录下的themes\sphinxdoc文件夹复制到theme文件夹下，并重命名为pydoc，目录结构如下图所示：



theme文件夹的结构

  - 编辑layout.html文件。此文件是一个模板，Sphinx最终使用此模板生成每个rst文件所对应的html文件。因此我在其中添加了对我自己的css和js文件的引用:

```
<link type="text/css" href="_static/jquery-ui-1.7.2.custom.css" rel
<link type="text/css" href="_static/comments.css" rel="stylesheet"
<script type="text/javascript" src="_static/jquery-ui-1.7.2.custom
<script type="text/javascript" src="_static/pydoc.js"></script>
```

- 在theme\pydoc\static目录下添加相应的css和js文件。为了固定html页面左侧的目录栏，可以配置theme\pydoc\theme.conf中的stickysidebar=True，不过好像IE7.0下无法正常显示，因此在css文件中添加如下代码，除了IE6.0以外其它的浏览器(Firefox,IE7, Chrome)都能够正常固定目录栏：

```
div.sphinxsidebar{
    position : fixed;
    left : 0px;
    top : 30px;
    margin-left : 0px !important;
}
```

关闭引号自动转换

在输出html的时候，如果使用Sphinx缺省的配置，会对引号进行自动转换：将标准的单引号和双引号转换为unicode中的全角引号。为了关闭此项功能，需要编辑conf.py，进行如下设置：

```
html_use_smartypants = False
```

用latex编写数学公式

Sphinx支持将latex编写的数学公式转换为png图片。为了在windows下使用latex，我下载了proTeXt，这个tex软件包的大小有700M左右，安装之后占用1.3G。为了告诉Sphinx工具latex的安装位置，如下修改make.bat文件：

```
%SPHINXBUILD% -D pngmath_latex="..\latex.exe" -b  html %ALLSPHINXOI
```

然后就可以如下使用latex:

```
X_k =  \sum_{n=0}^{N-1} x_n e^{-{i 2\pi k \frac{n}{N}}} \qquad k =
```

得到的输出图片如下：

$$X_k = \sum_{n=0}^{N-1} x_n e^{-{i 2\pi k \frac{n}{N}}} \qquad k = 0, \ldots, N-1.$$

Leo的配置

Leo的缺省配置用起来很不习惯：它的树状目录在上方，而且字体很小。下面是对Leo的一些修改和配置：

- Leo现在可以使用tk和qt两个库。使用tk库的界面用起来不习惯，因此通过在启动Leo时添加参数强制使用qt库的界面：launchLeo.py --gui=qt 。
- 我个人很喜欢微软雅黑的汉字字体，但是由于雅黑字体的英文不是等宽的，因此用它来编辑代码的话就很不爽了。于是上网找到了一个雅黑和Consolas的复合字体：

  > YaHei Mono字体下载地址： http://hyry.dip.jp/files/yahei_mono.7z

- 复制一份leo\config\leoSettings.leo到同一目录，改名为myLeoSettings.leo。用Leo编辑此文件，在目录树中找到节点：qtGui plugin-->@data qt-gui-plugin-style-sheet，修改此样式表中的字体的定义，使用新安装的Yahei Mono字体。

  > ```
  > QTextEdit#richTextEdit {
  >     ...
  >     font-family: Yahei Mono;
  >     font-size: 17px;
  >     ...
  > }
  > ```

- 修改@settings-->Window-->@string initial_split_orientation节点和@settings-->Window-->Options for new windows-->@strings[vertical,horizontal] initial_splitter_orientation节点的值为horizontal。这样目录树和编辑框就是左右分栏的了。

- 在Leo中用@auto-rst输出rst文件时，会自动的将目录树中的节点名转换为rst文件中的标题。在rst中标题都是由下划线标出的。下划线的长度要求和文本的长度一致。由于Leo采用unicode表示文本，因此汉字的长度为1，但是rst编译器似乎要求汉字的长度为2，因此对于 **Leo**的配置 这样的标题，rst要求用9个下划线符号标识，而Leo只用6个，造成在编译时出现许多警告信息，为了解决这个问题，编辑leo\core\leoRst.py文件中的underline函数如下，并且将其后的所有len(s)都改为len(ss)：

  ```
  def underline (self,s,p):
      ...

      try:
          ss = s.encode("gbk")
      except:
          try:
              ss = s.encode("shiftjis")
          except:
              ss = s

      trace = False and not g.unitTesting
      ...
  ```

让Matplotlib显示中文

将中文字体文件复制到：

```
%PythonPath%\Lib\site-packages\matplotlib\mpl-data\fonts\ttf\
```

下，这里以上一节介绍的Yahei Mono字体为例。

找到Matplotlib的配置文件matplotlibrc，全局配置文件的路径：

```
%PythonPath%\Lib\site-packages\matplotlib\mpl-data\matplotlibrc
```

用户配置文件路径：

```
c:\Documents and Settings\%UserName%\.matplotlib\matplotlibrc
```

用文本编辑器打开此文件，进行如下编辑：

- 找到设置font.family的行，改为font.family : monospace，注意去掉前面的#号。
- 在下面添加一行：font.monospace : Yahei Mono。

在matplotlib中使用中文字符串时记住要用unicode格式，例如：u"测试中文显示"。

用Matplotlib生成图片

matplotlib提供了一个Sphinx的扩展插件，可以使用..plot命令自动生成图片。可是这个插件生成的图片的路径和本书所采用的路径不符合，无法在HTML文件中显示最终生成的图。因此我直接复制下面两个文件：

```
c:\Python26\Lib\site-packages\matplotlib\sphinxext\plot_directive.p
c:\Python26\Lib\site-packages\matplotlib\sphinxext\only_directives.
```

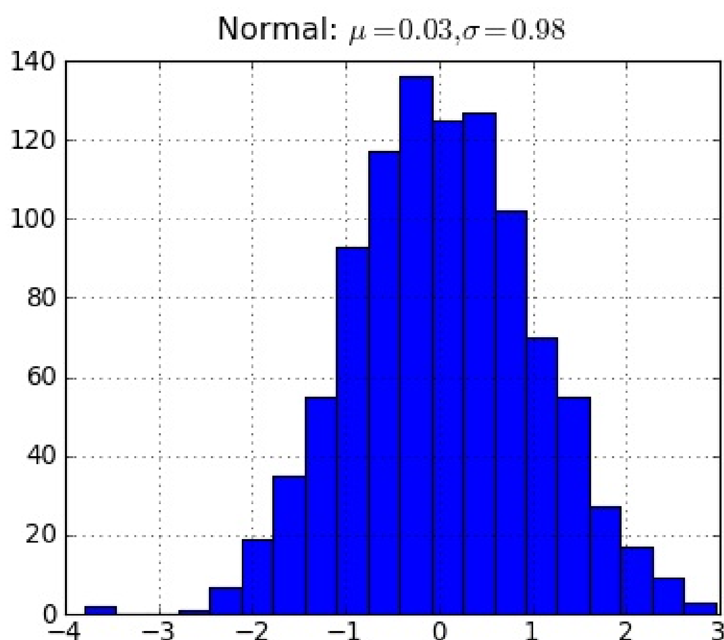
到sourceexts下，命名为plot_directive.py。然后编辑conf.py，修改下面两行：

```
sys.path.append(os.path.abspath('exts'))
extensions = ['sphinx.ext.autodoc', 'sphinx.ext.doctest',
    'sphinx.ext.pngmath', 'plot_directive']
```

这样就可以载入extsplot_directive.py扩展插件了。然后编辑plot_directive.py文件，使得它的输出符合本书的路径，并且除去大图和PDF输出。

在rst文件中使用：

```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.randn(1000)
plt.hist( x, 20)
plt.grid()
plt.title(r'Normal: $\mu=%.2f, \sigma=%.2f$'%(x.mean(), x.std()))
plt.show()
```



用Graphviz绘图

Sphinx可以调用Graphviz绘制流程图，首先下载Graphviz的Windows安装包进行安装，假设安装目录为c:\graphviz。

Graphviz的下载地址： http://www.graphviz.org

编辑conf.py配置文件，在 extensions 列表定义的最后添加一项：'sphinx.ext.graphviz'。

如下编辑make.bat文件，配置dot.exe的执行路径：

```
.. graphviz::

    digraph GraphvizDemo{
        node [fontname="Yahei Mono" shape="rect"];
        edge [fontname="Yahei Mono" fontsize=10];

        node1[label="开始"];
        node2[label="正常"];

        node1->node2[label="测试"];
    }
```

输出图为:

![digraph GraphvizDemo{ node [fontname="Yahei Mono" shape="rect"]; edge [fontname="Yahei Mono" fontsize=10];

```
node1[label="开始"];
node2[label="正常"];

node1-&gt;node2[label="测试"];
```

}](img/graphviz-691597b9de6125817b93aaad942bf30f1e3d5346.png)

制作CHM文档

Sphinx支持输出为CHM文档格式，只需要运行make htmlhelp即可。但是此命令输出的目录文件(扩展名为.hhc)，却不支持中文。为了解决这个问题，我进行了如下修改：

- sphinx的安装目录下找到buildershtmlhelp.py，将其复制一份，改名为htmlhelpcn.py。输出CHM文档的程序都在这里面。
- 修改builders__init__.py文件，在其最后的BUILTIN_BUILDERS字典定义中添加一行：

  ```
  'htmlhelpcn':  ('htmlhelpcn', 'HTMLHelpBuilder')
  ```

- 修改make.bat文件，在其中添加：

  ```
  if "%1" == "htmlhelpcn" (
      %SPHINXBUILD% -b htmlhelpcn %ALLSPHINXOPTS% build/htmlh
      echo.
      echo.Build finished; now you can run HTML Help Workshop
  .hhp project file in build/htmlhelpcn.
      goto end
  )
  ```

- 编辑htmlhelpcn.py文件，找到project_template字符串的定义，修改其中的Language定义为Language=0x804。
- 反复运行make.bat htmlhelpcn命令，根据输出的错误提示修改htmlhelpcn.py，将其中几处编码错误的地方都添加.encode("gb2312")。其中有一处：

  ```
  f.write(item.encode('ascii', 'xmlcharrefreplace'))

  # 改为--&gt;

  f.write(item.encode('gb2312'))
  ```

- 如果在rst文档中给图片添加了中文说明的话，有可能输出的CHM文件中看不到图片。
    - make.bat htmlhelpcn正常运行之后，运行下面的命令输出制作CHM文件：

```
"C:\Program Files\HTML Help Workshop\hhc.exe" htmlhelpcn\sc
```

CHM中嵌入Flash动画

用如下的reStructuredText的 raw 指令可以在html中嵌入Flash动画：

```
<OBJECT CLASSID="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" WIDTH=
  CODEBASE="http://active.macromedia.com/flash5/cabs/swflash.cab#ve
<PARAM NAME="movie" VALUE="img/fft_study_04.swf">
<PARAM NAME="play" VALUE="true">
<PARAM NAME="loop" VALUE="false">
<PARAM NAME="wmode" VALUE="transparent">
<PARAM NAME="quality" VALUE="high">
<EMBED SRC="img/fft_study_04.swf" width="589" HEIGHT="447" quality=
 loop="false" wmode="transparent" TYPE="application/x-shockwave-fla
 PLUGINSPAGE=
 "http://www.macromedia.com/shockwave/download/index.cgi?P1_Prod_Ve
</EMBED>
</OBJECT>
```

由于Html Help Workshop不会将swf文件打包进CHM，因此CHM中看不到flash动画，只需要在嵌入flash动画的html之后添加一条：

```
<img src="img/fft_study_04.swf" style="visibility:hidden"/>
```

这样Html Help Workshop就会把fft_study_04.swf文件添加进去，由于使用隐藏的CSS，页面中也不会把它当作图片显示出来。

制作PDF文档

调用make latex命令可以输出为latex格式的文件，然后调用 xelatex scipydoc.tex 即可将其转换为PDF文件，xelatex是proTeXt自带的命令。制作PDF文档时同样有中文无法显示的问题，按照以下步骤解决：

- 编辑文档的配置文件conf.py，在最后的 Options for LaTeX output 定义处，添加如下代码，这段文字将添加到最终输出的tex文件中，这里的Yahei Mono可以修改为你想要的中文字体名：

```
latex_preamble = r"""
 \usepackage{float}
 \textwidth 6.5in
 \oddsidemargin -0.2in
 \evensidemargin -0.2in
 \usepackage{ccaption}
 \usepackage{fontspec,xunicode,xltxtra}
 \setsansfont{Microsoft YaHei}
 \setromanfont{Microsoft YaHei}
 \setmainfont{Microsoft YaHei}
 \setmonofont{Yahei Mono}
 \XeTeXlinebreaklocale "zh"
 \XeTeXlinebreakskip = 0pt plus 1pt
 \renewcommand{\baselinestretch}{1.3}
 \setcounter{tocdepth}{3}
 \captiontitlefont{\small\sffamily}
 \captiondelim{ - }
 \renewcommand\today{\number\year年\number\month月\number\day日}
 \makeatletter
 \renewcommand*\l@subsection{\@dottedtocline{2}{2.0em}{4.0em}}
 \renewcommand*\l@subsubsection{\@dottedtocline{3}{3em}{5em}}
 \makeatother
 \titleformat{\chapter}[display]
 {\bfseries\Huge}
 {\filleft \Huge 第 \hspace{2 mm} \thechapter \hspace{4 mm} 章}
 {4ex}
 {\titlerule
 \vspace{2ex}%
 \filright}
 [\vspace{2ex}%
 \titlerule]
 %\definecolor{VerbatimBorderColor}{rgb}{0.2,0.2,0.2}
 \definecolor{VerbatimColor}{rgb}{0.95,0.95,0.95}
""".decode("utf-8")
```

通过renewcommand命令将输出的PDF文档中的一部分英文修改为中文。

不知何故，在latex_preamble中添加修改插图标题前缀的命令没有作用，因此通过下面的命令在正文中添加转换前缀的renewcommand：

```
.. raw:: latex

    \renewcommand\partname{部分}
    \renewcommand{\chaptermark}[1]{\markboth{第 \thechapter\ 章 \hs
    \fancyhead[LE,RO]{用Python做科学计算}
    \renewcommand{\figurename}{\textsc{图}}
```

- 调整conf.py中的其它选项：

```
latex_paper_size = 'a4'
latex_font_size = '11pt'
latex_use_modindex = False
```

- 运行下面的命令输出PDF文档，使用nonstopmode，即使出现错误也不暂停运行。

```
xelatex -interaction=nonstopmode scipydoc.tex
```

还有一些latex配置没有找到如何使用reStructuredText进行设置，因此写了一个Python的小程序读取输出的tex文件，替换其中的一些latex命令：

- 将begin{figure}[htbp]改为begin{figure}[H]，这样能保证图和文字保持tex中的前后关系，而不会对图进行自动排版
- 在\tableofcontents之前添加\renewcommand\contentsname{目 录}，将目录标题的英文改为中文，此段配置在latex_preamble中定义无效

添加PDF封面

使用作图软件设计封面图片之后，使用图片转PDF工具将其转换为一个只有一页的PDF文档cover.pdf：

图片转PDF工具下载地址： http://www.softinterface.com

然后使用PDF合并工具将cover.pdf和正文的PDF文件进行合并。我在网络上找了很久，终于找到了下面这个能够维持内部链接和书签的免费的合并工具：

PDF工具PDFsam下载地址： http://www.pdfsam.org

PDFsam提供了界面和命令行方式，界面方式很容易使用，但是为了一个批处理产生最终PDF文档我需要使用命令行方式，下面是使用命令行进行PDF文档合并的批处理程序：

```
set MERGE=java -jar "c:\Program Files\pdfsam\lib\pdfsam-console-2.2
%MERGE% -f cover.pdf -f scipydoc.pdf -o %CD%\scipydoc2.pdf concat
```

- -f参数指定输入的PDF文件名
- -o参数指定输出的PDF文件名，注意必须使用绝对路径，因此这里使用%CD%将相对路径转换为绝对路径。

输出打包的批处理

下面是同时输出zip, chm, pdf文件的批处理命令：

```
rename html scipydoc
"c:\Program Files\7-Zip\7z.exe" a scipydoc.zip scipydoc
rename scipydoc html

"C:\Program Files\HTML Help Workshop\hhc.exe" htmlhelpcn\scipydoc.h
copy htmlhelpcn\scipydoc.chm . /y

cd latex
xelatex -interaction=nonstopmode scipydoc.tex
cd ..
copy latex\scipydoc.pdf . /y
```

HTML的中文搜索

由于Sphinx不懂中文分词，因此它所生成的搜索索引文件searchindex.js中的中文单词分的不正确。为了修正这个问题，我写了一个Sphinx扩展chinese_search.py，使用中文分词库smallseg生成索引文件中的中文单词。

smallseg中文分词库地址: http://code.google.com/p/smallseg

下面是这个扩展的完整源程序：

```
from os import path

import re
import cPickle as pickle

from docutils.nodes import comment, Text, NodeVisitor, SkipNode

from sphinx.util.stemmer import PorterStemmer
from sphinx.util import jsdump, rpartition

from smallseg import SEG

DEBUG = False

word_re = re.compile(r'\w+(?u)')

stopwords = set("""
a  and  are  as  at
be  but  by
for
if  in  into  is  it
near  no  not
of  on  or
such
that  the  their  then  there  these  they  this  to
was  will  with
""".split())
```

```
if DEBUG:
    testfile = file("testfile.txt", "wb")

class _JavaScriptIndex(object):
    """
 The search index as javascript file that calls a function
 on the documentation search object to register the index.
 """

    PREFIX = 'Search.setIndex('
    SUFFIX = ')'

    def dumps(self, data):
        return self.PREFIX + jsdump.dumps(data) + self.SUFFIX

    def loads(self, s):
        data = s[len(self.PREFIX):-len(self.SUFFIX)]
        if not data or not s.startswith(self.PREFIX) or not \
           s.endswith(self.SUFFIX):
            raise ValueError('invalid data')
        return jsdump.loads(data)

    def dump(self, data, f):
        f.write(self.dumps(data))

    def load(self, f):
        return self.loads(f.read())

js_index = _JavaScriptIndex()

class Stemmer(PorterStemmer):
    """
 All those porter stemmer implementations look hideous.
 make at least the stem method nicer.
 """

    def stem(self, word):
        word = word.lower()
        return word
        #return PorterStemmer.stem(self, word, 0, len(word) - 1)

class WordCollector(NodeVisitor):
    """
 A special visitor that collects words for the `IndexBuilder`.
 """

    def __init__(self, document):
        NodeVisitor.__init__(self, document)
        self.found_words = []

    def dispatch_visit(self, node):
        if node.__class__ is comment:
```

```
            raise SkipNode
        if node.__class__ is Text:
            words = seg.cut(node.astext().encode("utf8"))
            words.reverse()
            self.found_words.extend(words)

class IndexBuilder(object):
    """
 Helper class that creates a searchindex based on the doctrees
 passed to the `feed` method.
 """
    formats = {
        'jsdump':   jsdump,
        'pickle':   pickle
    }

    def __init__(self, env):
        self.env = env
        self._stemmer = Stemmer()
        # filename -> title
        self._titles = {}
        # stemmed word -> set(filenames)
        self._mapping = {}
        # desctypes -> index
        self._desctypes = {}

    def load(self, stream, format):
        """Reconstruct from frozen data."""
        if isinstance(format, basestring):
            format = self.formats[format]
        frozen = format.load(stream)
        # if an old index is present, we treat it as not existing.
        if not isinstance(frozen, dict):
            raise ValueError('old format')
        index2fn = frozen['filenames']
        self._titles = dict(zip(index2fn, frozen['titles']))
        self._mapping = {}
        for k, v in frozen['terms'].iteritems():
            if isinstance(v, int):
                self._mapping[k] = set([index2fn[v]])
            else:
                self._mapping[k] = set(index2fn[i] for i in v)
        # no need to load keywords/desctypes

    def dump(self, stream, format):
        """Dump the frozen index to a stream."""
        if isinstance(format, basestring):
            format = self.formats[format]
        format.dump(self.freeze(), stream)

    def get_modules(self, fn2index):
        rv = {}
        for name, (doc, _, _, _) in self.env.modules.iteritems():
```

```
            if doc in fn2index:
                rv[name] = fn2index[doc]
        return rv

    def get_descrefs(self, fn2index):
        rv = {}
        dt = self._desctypes
        for fullname, (doc, desctype) in self.env.descrefs.iteriter
            if doc not in fn2index:
                continue
            prefix, name = rpartition(fullname, '.')
            pdict = rv.setdefault(prefix, {})
            try:
                i = dt[desctype]
            except KeyError:
                i = len(dt)
                dt[desctype] = i
            pdict[name] = (fn2index[doc], i)
        return rv

    def get_terms(self, fn2index):
        rv = {}
        for k, v in self._mapping.iteritems():
            if len(v) == 1:
                fn, = v
                if fn in fn2index:
                    rv[k] = fn2index[fn]
            else:
                rv[k] = [fn2index[fn] for fn in v if fn in fn2index
        return rv

    def freeze(self):
        """Create a usable data structure for serializing."""
        filenames = self._titles.keys()
        titles = self._titles.values()
        fn2index = dict((f, i) for (i, f) in enumerate(filenames))
        return dict(
            filenames=filenames,
            titles=titles,
            terms=self.get_terms(fn2index),
            descrefs=self.get_descrefs(fn2index),
            modules=self.get_modules(fn2index),
            desctypes=dict((v, k) for (k, v) in self._desctypes.ite
        )

    def prune(self, filenames):
        """Remove data for all filenames not in the list."""
        new_titles = {}
        for filename in filenames:
            if filename in self._titles:
                new_titles[filename] = self._titles[filename]
        self._titles = new_titles
        for wordnames in self._mapping.itervalues():
```

```
                wordnames.intersection_update(filenames)

        def feed(self, filename, title, doctree):
            """Feed a doctree to the index."""
            self._titles[filename] = title

            visitor = WordCollector(doctree)
            doctree.walk(visitor)

            def add_term(word, prefix='', stem=self._stemmer.stem):
                word = stem(word)
                word = word.strip(u"!@#$%^&*()_+-*/\\\";,.[]{}<>")
                if len(word) <= 1: return
                if word.encode("utf8").isalpha() and len(word) < 3: ret
                if word.isdigit(): return
                if word in stopwords: return

                try:
                    float(word)
                    return
                except:
                    pass

                if DEBUG:
                    testfile.write("%s\n" % word.encode("utf8"))
                self._mapping.setdefault(prefix + word, set()).add(file

            words = seg.cut(title.encode("utf8"))
            for word in words:
                add_term(word)

            for word in visitor.found_words:
                add_term(word)

    def load_indexer(self):
        def func(docnames):
            print "############### CHINESE INDEXER ###############"
            self.indexer = IndexBuilder(self.env)
            keep = set(self.env.all_docs) - set(docnames)
            try:
                f = open(path.join(self.outdir, self.searchindex_filena
                try:
                    self.indexer.load(f, self.indexer_format)
                finally:
                    f.close()
            except (IOError, OSError, ValueError):
                if keep:
                    self.warn('search index couldn\'t be loaded, but no
                              'documents will be built: the index will
                              'incomplete.')
            # delete all entries for files that will be rebuilt
            self.indexer.prune(keep)
        return func
```

```
def builder_inited(app):
    if app.builder.name == 'html':
        print "*****************************"
        global seg
        seg = SEG()
        app.builder.load_indexer = load_indexer(app.builder)

def setup(app):
    app.connect('builder-inited', builder_inited)
```

PDF的页码和图编号参照

Sphinx生成的tex文件没有使用\label和\ref进行编号引用，而是生成一些链接，这些链接虽然方便电子版的阅读，可是打印出来之后就毫无用处了，因此我写了一个扩展latex_ref.py为最终生成的PDF添加编号引用功能，这个扩展添加了三个role：tlabel, tref, tpageref，分别对应tex的\label, \ref, \pageref。

下面是完整的源程序：

```
# -*- coding: utf-8 -*-
from docutils import nodes, utils

class tref(nodes.Inline, nodes.TextElement):
    pass

class tlabel(nodes.Inline, nodes.TextElement):
    pass

class tpageref(nodes.Inline, nodes.TextElement):
    pass

def tref_role(role, rawtext, text, lineno, inliner, options={}, con
    data = text.split(",")
    if u"图" in data[0]:
        name = u"图"
        pos = data[0][0]
        ref = data[1]
        return [tref(name=name, ref=ref, pos=pos)], []
    return [],[]
def tlabel_role(role, rawtext, text, lineno, inliner, options={}, c
    return [tlabel(latex=text)], []

def tpageref_role(role, rawtext, text, lineno, inliner, options={},
    return [tpageref(latex=text)], []

def latex_visit_ref(self, node):
    self.body.append(r"%s\ref{%s}" % (node['name'], node['ref']))
    raise nodes.SkipNode
```

```
def html_visit_ref(self, node):
    self.body.append(r'<a href="#%s">%s%s</a>' % (node['ref'], node
    raise nodes.SkipNode

def latex_visit_label(self, node):
    self.body.append(r"\label{%s}" % node['latex'])
    raise nodes.SkipNode

def latex_visit_pageref(self, node):
    self.body.append(r"\pageref{%s}" % node['latex'])
    raise nodes.SkipNode

def empty_visit(self, node):
    raise nodes.SkipNode

def setup(app):
    app.add_node(tref,latex=(latex_visit_ref, None),text=(empty_vis
    app.add_node(tlabel,latex=(latex_visit_label, None),text=(empty
    app.add_node(tpageref,latex=(latex_visit_pageref, None),text=(e

    app.add_role('tref', tref_role)
    app.add_role('tlabel', tlabel_role)
    app.add_role('tpageref', tpageref_role)
```

ReST使用心得

添加图的编号和标题

使用figure命令插入带编号和标题的插图：

```
.. _pythonxyhome:

.. figure:: images/pythonxy_home.png

    Python(x,y)的启动画面
```

PDF文字包围图片

当给figure添加figwidth和align属性之后，在生成的latex文档中，将使用wrapfigure生成图。为了和前面的段落之间添加一个换行符，使用一个斜杠空格。

```
.. literalinclude:: examples/tvtk_cone.example.py

.. literalinclude:: example.c
    :language: c
```

未解决的问题

数学公式输出不正确

有时候数学公式的输出不正确，某些数学符号不能显示，可是多试几次之后就正常了，不知道是什么原因。

Leo不能配置目录树和编辑框的宽度比例

每次Leo开启之后目录树和编辑框的宽度是相等的，看上去很不协调。而且修改mySettings.leo中的相关配置也不能解决，不明白是什么问题。目前的解决方法是添加两个工具按钮：show-tree和hide-tree，这样点击一下show-tree就会将目录树和编辑框改为1:3的比例；而点击hide-tree则能隐藏目录树：

```
# -*- coding: utf-8 -*-
from enthought.traits.api import \
    Str, Float, HasTraits, Property, cached_property, Range, Instar

from enthought.chaco.api import Plot, AbstractPlotData, ArrayPlotDa

from enthought.traits.ui.api import \
    Item, View, VGroup, HSplit, ScrubberEditor, VSplit

from enthought.enable.api import Component, ComponentEditor
from enthought.chaco.tools.api import PanTool, ZoomTool

import numpy as np

# 鼠标拖动修改值的控件的样式
scrubber = ScrubberEditor(
    hover_color  = 0xFFFFFF,
    active_color = 0xA0CD9E,
    border_color = 0x808080
)

# 取FFT计算的结果freqs中的前n项进行合成，返回合成结果，计算loops个周期的波形
def fft_combine(freqs, n, loops=1):
    length = len(freqs) * loops
    data = np.zeros(length)
    index = loops * np.arange(0, length, 1.0) / length * (2 * np.pi
    for k, p in enumerate(freqs[:n]):
        if k != 0: p *= 2 # 除去直流成分之外，其余的系数都*2
        data += np.real(p) * np.cos(k*index) # 余弦成分的系数为实数部
        data -= np.imag(p) * np.sin(k*index) # 正弦成分的系数为负的虚数
    return index, data

class TriangleWave(HasTraits):
    # 指定三角波的最窄和最宽范围，由于Range似乎不能将常数和traits名混用
    # 所以定义这两个不变的trait属性
    low = Float(0.02)
    hi = Float(1.0)
```

```
    # 三角波形的宽度
    wave_width = Range("low", "hi", 0.5)

    # 三角波的顶点C的x轴坐标
    length_c = Range("low", "wave_width", 0.5)

    # 三角波的定点的y轴坐标
    height_c = Float(1.0)

    # FFT计算所使用的取样点数，这里用一个Enum类型的属性以供用户从列表中选择
    fftsize = Enum( [(2**x) for x in range(6, 12)])

    # FFT频谱图的x轴上限值
    fft_graph_up_limit = Range(0, 400, 20)

    # 用于显示FFT的结果
    peak_list = Str

    # 采用多少个频率合成三角波
    N = Range(1, 40, 4)

    # 保存绘图数据的对象
    plot_data = Instance(AbstractPlotData)

    # 绘制波形图的容器
    plot_wave = Instance(Component)

    # 绘制FFT频谱图的容器
    plot_fft  = Instance(Component)

    # 包括两个绘图的容器
    container = Instance(Component)

    # 设置用户界面的视图， 注意一定要指定窗口的大小，这样绘图容器才能正常初始化
    view = View(
        HSplit(
            VSplit(
                VGroup(
                    Item("wave_width", editor = scrubber, label=u"波
                    Item("length_c", editor = scrubber, label=u"最高
                    Item("height_c", editor = scrubber, label=u"最高
                    Item("fft_graph_up_limit", editor = scrubber, l
                    Item("fftsize", label=u"FFT点数"),
                    Item("N", label=u"合成波频率数")
                ),
                Item("peak_list", style="custom", show_label=False,
            ),
            VGroup(
                Item("container", editor=ComponentEditor(size=(600,
                orientation = "vertical"
            )
        ),
```

```
        resizable = True,
        width = 800,
        height = 600,
        title = u"三角波FFT演示"
    )

    # 创建绘图的辅助函数，创建波形图和频谱图有很多类似的地方，因此单独用一个函数
    # 减少重复代码
    def _create_plot(self, data, name, type="line"):
        p = Plot(self.plot_data)
        p.plot(data, name=name, title=name, type=type)
        p.tools.append(PanTool(p))
        zoom = ZoomTool(component=p, tool_mode="box", always_on=Fal
        p.overlays.append(zoom)
        p.title = name
        return p

    def __init__(self):
        # 首先需要调用父类的初始化函数
        super(TriangleWave, self).__init__()

        # 创建绘图数据集，暂时没有数据因此都赋值为空，只是创建几个名字，以供Pl
        self.plot_data = ArrayPlotData(x=[], y=[], f=[], p=[], x2=[

        # 创建一个垂直排列的绘图容器，它将频谱图和波形图上下排列
        self.container = VPlotContainer()

        # 创建波形图，波形图绘制两条曲线： 原始波形(x,y)和合成波形(x2,y2)
        self.plot_wave = self._create_plot(("x","y"), "Triangle Wav
        self.plot_wave.plot(("x2","y2"), color="red")

        # 创建频谱图，使用数据集中的f和p
        self.plot_fft  = self._create_plot(("f","p"), "FFT", type="

        # 将两个绘图容器添加到垂直容器中
        self.container.add( self.plot_wave )
        self.container.add( self.plot_fft )

        # 设置
        self.plot_wave.x_axis.title = "Samples"
        self.plot_fft.x_axis.title = "Frequency pins"
        self.plot_fft.y_axis.title = "(dB)"

        # 改变fftsize为1024，因为Enum的默认缺省值为枚举列表中的第一个值
        self.fftsize = 1024

    # FFT频谱图的x轴上限值的改变事件处理函数，将最新的值赋值给频谱图的响应属性
    def _fft_graph_up_limit_changed(self):
        self.plot_fft.x_axis.mapper.range.high = self.fft_graph_up_

    def _N_changed(self):
        self.plot_sin_combine()
```

```
    # 多个trait属性的改变事件处理函数相同时，可以用@on_trait_change指定
    @on_trait_change("wave_width, length_c, height_c, fftsize")
    def update_plot(self):
        # 计算三角波
        global y_data
        x_data = np.arange(0, 1.0, 1.0/self.fftsize)
        func = self.triangle_func()
        # 将func函数的返回值强制转换成float64
        y_data = np.cast["float64"](func(x_data))

        # 计算频谱
        fft_parameters = np.fft.fft(y_data) / len(y_data)

        # 计算各个频率的振幅
        fft_data = np.clip(20*np.log10(np.abs(fft_parameters))[:sel

        # 将计算的结果写进数据集
        self.plot_data.set_data("x", np.arange(0, self.fftsize)) #
        self.plot_data.set_data("y", y_data)
        self.plot_data.set_data("f", np.arange(0, len(fft_data))) #
        self.plot_data.set_data("p", fft_data)

        # 合成波的x坐标为取样点，显示2个周期
        self.plot_data.set_data("x2", np.arange(0, 2*self.fftsize)

        # 更新频谱图x轴上限
        self._fft_graph_up_limit_changed()

        # 将振幅大于-80dB的频率输出
        peak_index = (fft_data > -80)
        peak_value = fft_data[peak_index][:20]
        result = []
        for f, v in zip(np.flatnonzero(peak_index), peak_value):
            result.append("%s : %s" %(f, v) )
        self.peak_list = "\n".join(result)

        # 保存现在的fft计算结果，并计算正弦合成波
        self.fft_parameters = fft_parameters
        self.plot_sin_combine()

    # 计算正弦合成波，计算2个周期
    def plot_sin_combine(self):
        index, data = fft_combine(self.fft_parameters, self.N, 2)
        self.plot_data.set_data("y2", data)

    # 返回一个ufunc计算指定参数的三角波
    def triangle_func(self):
        c = self.wave_width
        c0 = self.length_c
        hc = self.height_c

        def trifunc(x):
            x = x - int(x) # 三角波的周期为1，因此只取x坐标的小数部分进行
```

```
            if x >= c: r = 0.0
            elif x < c0: r = x / c0 * hc
            else: r = (c-x) / (c-c0) * hc
            return r

        # 用trifunc函数创建一个ufunc函数，可以直接对数组进行计算, 不过通过此
        # 计算得到的是一个Object数组，需要进行类型转换
        return np.frompyfunc(trifunc, 1, 1)

if __name__ == "__main__":
    triangle = TriangleWave()
    triangle.configure_traits()
```

最近更新

- 2010/01/15: 将*Mayavi*嵌入到界面中
- 2010/01/14: 模拟*IIR*滤波器的频带转换
- 2010/01/12: 修改Sphinx模板，添加支持中文搜索的插件，中文分词库采用

  > smallseg： http://code.google.com/p/smallseg

- 2010/01/07: 巴特沃斯低通滤波器 ； 双线性变换
- 2010/01/05: 用*Sympy*计算球体体积 ； *NumPy*-快速处理数据 添加少许新内容；修改章节名
- 2010/01/04: *L-System*分形
- 2010/01/03: 迭代函数系统设计器
- 2010/01/02: 迭代函数系统*(IFS)*
- 2009/12/30 : *Matplotlib*的*Axis*对象
- 2009/12/29 : 绘制*Mandelbrot*集合

源程序集

- 三角波的FFT演示
- 在traitsUI中使用的matplotlib控件
- CSV文件数据图形化工具
- NLMS算法的模拟测试
- 三维标量场观察器
- 频谱泄漏和hann窗
- FFT卷积的速度比较
- 二次均衡器设计
- 单摆摆动周期的计算
- 双摆系统的动画模拟
- 绘制Mandelbrot集合
- 迭代函数系统的分形
- 绘制L-System的分形图

三角波的FFT演示

相关文档： *FFT演示程序*

本程序演示各种三角波形的FFT频谱，用户可以方便地修改三角波的各个参数，并立即看到其FFT频谱的变化。



```
# -*- coding: utf-8 -*-
from enthought.traits.api import \
    Str, Float, HasTraits, Property, cached_property, Range, Instar

from enthought.chaco.api import Plot, AbstractPlotData, ArrayPlotDa

from enthought.traits.ui.api import \
    Item, View, VGroup, HSplit, ScrubberEditor, VSplit

from enthought.enable.api import Component, ComponentEditor
from enthought.chaco.tools.api import PanTool, ZoomTool

import numpy as np

# 鼠标拖动修改值的控件的样式
scrubber = ScrubberEditor(
    hover_color  = 0xFFFFFF,
    active_color = 0xA0CD9E,
    border_color = 0x808080
)

# 取FFT计算的结果freqs中的前n项进行合成，返回合成结果，计算loops个周期的波形
def fft_combine(freqs, n, loops=1):
    length = len(freqs) * loops
    data = np.zeros(length)
    index = loops * np.arange(0, length, 1.0) / length * (2 * np.pi
    for k, p in enumerate(freqs[:n]):
        if k != 0: p *= 2 # 除去直流成分之外，其余的系数都*2
        data += np.real(p) * np.cos(k*index) # 余弦成分的系数为实数部
        data -= np.imag(p) * np.sin(k*index) # 正弦成分的系数为负的虚数
    return index, data

class TriangleWave(HasTraits):
    # 指定三角波的最窄和最宽范围，由于Range似乎不能将常数和traits名混用
    # 所以定义这两个不变的trait属性
    low = Float(0.02)
    hi = Float(1.0)

    # 三角波形的宽度
    wave_width = Range("low", "hi", 0.5)
```

```
    # 三角波的顶点C的x轴坐标
    length_c = Range("low", "wave_width", 0.5)

    # 三角波的定点的y轴坐标
    height_c = Float(1.0)

    # FFT计算所使用的取样点数，这里用一个Enum类型的属性以供用户从列表中选择
    fftsize = Enum( [(2**x) for x in range(6, 12)])

    # FFT频谱图的x轴上限值
    fft_graph_up_limit = Range(0, 400, 20)

    # 用于显示FFT的结果
    peak_list = Str

    # 采用多少个频率合成三角波
    N = Range(1, 40, 4)

    # 保存绘图数据的对象
    plot_data = Instance(AbstractPlotData)

    # 绘制波形图的容器
    plot_wave = Instance(Component)

    # 绘制FFT频谱图的容器
    plot_fft  = Instance(Component)

    # 包括两个绘图的容器
    container = Instance(Component)

    # 设置用户界面的视图， 注意一定要指定窗口的大小，这样绘图容器才能正常初始化
    view = View(
        HSplit(
            VSplit(
                VGroup(
                    Item("wave_width", editor = scrubber, label=u"波
                    Item("length_c", editor = scrubber, label=u"最高
                    Item("height_c", editor = scrubber, label=u"最高
                    Item("fft_graph_up_limit", editor = scrubber, l
                    Item("fftsize", label=u"FFT点数"),
                    Item("N", label=u"合成波频率数")
                ),
                Item("peak_list", style="custom", show_label=False,
            ),
            VGroup(
                Item("container", editor=ComponentEditor(size=(600,
                orientation = "vertical"
            )
        ),
        resizable = True,
        width = 800,
        height = 600,
        title = u"三角波FFT演示"
```

```
    )

    # 创建绘图的辅助函数，创建波形图和频谱图有很多类似的地方，因此单独用一个函数
    # 减少重复代码
    def _create_plot(self, data, name, type="line"):
        p = Plot(self.plot_data)
        p.plot(data, name=name, title=name, type=type)
        p.tools.append(PanTool(p))
        zoom = ZoomTool(component=p, tool_mode="box", always_on=Fal
        p.overlays.append(zoom)
        p.title = name
        return p

    def __init__(self):
        # 首先需要调用父类的初始化函数
        super(TriangleWave, self).__init__()

        # 创建绘图数据集，暂时没有数据因此都赋值为空，只是创建几个名字，以供Pl
        self.plot_data = ArrayPlotData(x=[], y=[], f=[], p=[], x2=[

        # 创建一个垂直排列的绘图容器，它将频谱图和波形图上下排列
        self.container = VPlotContainer()

        # 创建波形图，波形图绘制两条曲线： 原始波形(x,y)和合成波形(x2,y2)
        self.plot_wave = self._create_plot(("x","y"), "Triangle Wav
        self.plot_wave.plot(("x2","y2"), color="red")

        # 创建频谱图，使用数据集中的f和p
        self.plot_fft  = self._create_plot(("f","p"), "FFT", type="

        # 将两个绘图容器添加到垂直容器中
        self.container.add( self.plot_wave )
        self.container.add( self.plot_fft )

        # 设置
        self.plot_wave.x_axis.title = "Samples"
        self.plot_fft.x_axis.title = "Frequency pins"
        self.plot_fft.y_axis.title = "(dB)"

        # 改变fftsize为1024，因为Enum的默认缺省值为枚举列表中的第一个值
        self.fftsize = 1024

    # FFT频谱图的x轴上限值的改变事件处理函数，将最新的值赋值给频谱图的响应属性
    def _fft_graph_up_limit_changed(self):
        self.plot_fft.x_axis.mapper.range.high = self.fft_graph_up_

    def _N_changed(self):
        self.plot_sin_combine()

    # 多个trait属性的改变事件处理函数相同时，可以用@on_trait_change指定
    @on_trait_change("wave_width, length_c, height_c, fftsize")
    def update_plot(self):
        # 计算三角波
```

```
        global y_data
        x_data = np.arange(0, 1.0, 1.0/self.fftsize)
        func = self.triangle_func()
        # 将func函数的返回值强制转换成float64
        y_data = np.cast["float64"](func(x_data))

        # 计算频谱
        fft_parameters = np.fft.fft(y_data) / len(y_data)

        # 计算各个频率的振幅
        fft_data = np.clip(20*np.log10(np.abs(fft_parameters))[:se

        # 将计算的结果写进数据集
        self.plot_data.set_data("x", np.arange(0, self.fftsize)) #
        self.plot_data.set_data("y", y_data)
        self.plot_data.set_data("f", np.arange(0, len(fft_data))) #
        self.plot_data.set_data("p", fft_data)

        # 合成波的x坐标为取样点，显示2个周期
        self.plot_data.set_data("x2", np.arange(0, 2*self.fftsize)

        # 更新频谱图x轴上限
        self._fft_graph_up_limit_changed()

        # 将振幅大于-80dB的频率输出
        peak_index = (fft_data > -80)
        peak_value = fft_data[peak_index][:20]
        result = []
        for f, v in zip(np.flatnonzero(peak_index), peak_value):
            result.append("%s : %s" %(f, v) )
        self.peak_list = "\n".join(result)

        # 保存现在的fft计算结果，并计算正弦合成波
        self.fft_parameters = fft_parameters
        self.plot_sin_combine()

    # 计算正弦合成波，计算2个周期
    def plot_sin_combine(self):
        index, data = fft_combine(self.fft_parameters, self.N, 2)
        self.plot_data.set_data("y2", data)

    # 返回一个ufunc计算指定参数的三角波
    def triangle_func(self):
        c = self.wave_width
        c0 = self.length_c
        hc = self.height_c

        def trifunc(x):
            x = x - int(x) # 三角波的周期为1，因此只取x坐标的小数部分进行
            if x >= c: r = 0.0
            elif x < c0: r = x / c0 * hc
            else: r = (c-x) / (c-c0) * hc
            return r
```

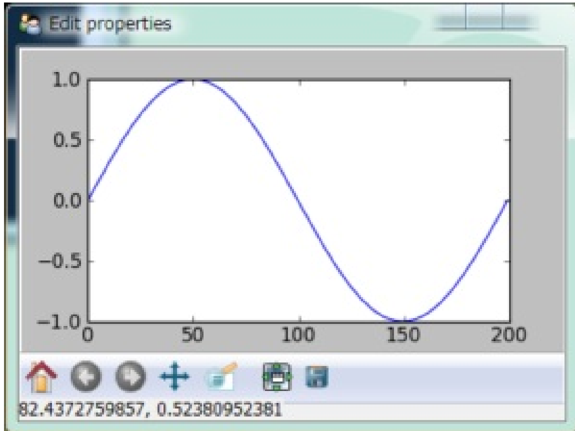
```
        # 用trifunc函数创建一个ufunc函数，可以直接对数组进行计算，不过通过此
        # 计算得到的是一个Object数组，需要进行类型转换
        return np.frompyfunc(trifunc, 1, 1)

if __name__ == "__main__":
    triangle = TriangleWave()
    triangle.configure_traits()
```

在traitsUI中使用的matplotlib控件

相关文档： 设计自己的*Trait*编辑器

在traitsUI所产生的界面中嵌入matplotlib的控件。




```
# -*- coding: utf-8 -*-
# file name: mpl_figure_editor.py
import wx
import matplotlib
# matplotlib采用WXAgg为后台，这样才能将绘图控件嵌入以wx为后台界面库的traits
matplotlib.use("WXAgg")
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as
from matplotlib.backends.backend_wx import NavigationToolbar2Wx
from enthought.traits.ui.wx.editor import Editor
from enthought.traits.ui.basic_editor_factory import BasicEditorFac

class _MPLFigureEditor(Editor):
    """
 相当于wx后台界面库中的编辑器，它负责创建真正的控件
 """
    scrollable = True

    def init(self, parent):
        self.control = self._create_canvas(parent)
        self.set_tooltip()
        print dir(self.item)

    def update_editor(self):
        pass

    def _create_canvas(self, parent):
        """
 创建一个Panel, 布局采用垂直排列的BoxSizer, panel中中添加
 FigureCanvas, NavigationToolbar2Wx, StaticText三个控件
 FigureCanvas的鼠标移动事件调用mousemoved函数，在StaticText
 显示鼠标所在的数据坐标
```

```
"""
        panel = wx.Panel(parent, -1, style=wx.CLIP_CHILDREN)
        def mousemoved(event):
            panel.info.SetLabel("%s, %s" % (event.xdata, event.ydat
        panel.mousemoved = mousemoved
        sizer = wx.BoxSizer(wx.VERTICAL)
        panel.SetSizer(sizer)
        mpl_control = FigureCanvas(panel, -1, self.value)
        mpl_control.mpl_connect("motion_notify_event", mousemoved)
        toolbar = NavigationToolbar2Wx(mpl_control)
        sizer.Add(mpl_control, 1, wx.LEFT | wx.TOP | wx.GROW)
        sizer.Add(toolbar, 0, wx.EXPAND|wx.RIGHT)
        panel.info = wx.StaticText(parent, -1)
        sizer.Add(panel.info)

        self.value.canvas.SetMinSize((10,10))
        return panel

class MPLFigureEditor(BasicEditorFactory):
    """
 相当于traits.ui中的EditorFactory，它返回真正创建控件的类
"""
    klass = _MPLFigureEditor

if __name__ == "__main__":
    from matplotlib.figure import Figure
    from enthought.traits.api import HasTraits, Instance
    from enthought.traits.ui.api import View, Item
    from numpy import sin, cos, linspace, pi

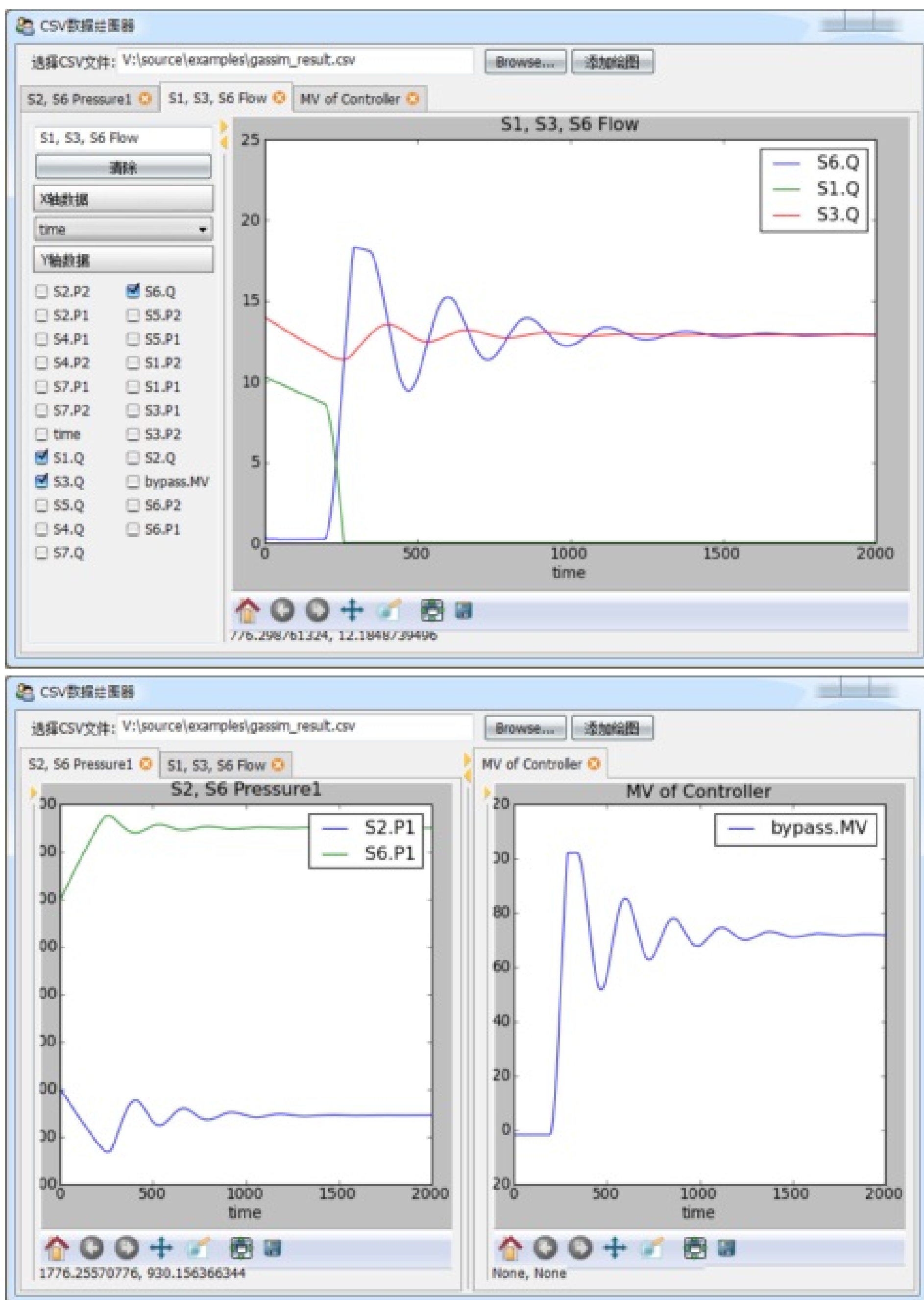
    class Test(HasTraits):
        figure = Instance(Figure, ())
        view = View(
            Item("figure", editor=MPLFigureEditor(), show_label=Fal
            width = 400,
            height = 300,
            resizable = True)
        def __init__(self):
            super(Test, self).__init__()
            axes = self.figure.add_subplot(111)
            t = linspace(0, 2*pi, 200)
            axes.plot(sin(t))

    Test().configure_traits()
```

CSV文件数据图形化工具

相关文档： 设计自己的*Trait*编辑器

采用 在*traitsUI*中使用的*matplotlib*控件 制作的CSV文件数据绘图工具。




```
# -*- coding: utf-8 -*-
from matplotlib.figure import Figure
from mpl_figure_editor import MPLFigureEditor
from enthought.traits.ui.api import *
from enthought.traits.api import *
import csv
```

```
class DataSource(HasTraits):
    """
 数据源，data是一个字典，将字符串映射到列表
 names是data中的所有字符串的列表
 """
    data = DictStrAny
    names = List(Str)

    def load_csv(self, filename):
        """
 从CSV文件读入数据，更新data和names属性
 """
        f = file(filename)
        reader = csv.DictReader(f)
        self.names = reader.fieldnames
        for field in reader.fieldnames:
            self.data[field] = []
        for line in reader:
            for k, v in line.iteritems():
                self.data[k].append(float(v))
        f.close()

class Graph(HasTraits):
    """
 绘图组件，包括左边的数据选择控件和右边的绘图控件
 """
    name = Str # 绘图名，显示在标签页标题和绘图标题中
    data_source = Instance(DataSource) # 保存数据的数据源
    figure = Instance(Figure) # 控制绘图控件的Figure对象
    selected_xaxis = Str # X轴所用的数据名
    selected_items = List # Y轴所用的数据列表

    clear_button = Button(u"清除") # 快速清除Y轴的所有选择的数据

    view = View(
        HSplit( # HSplit分为左右两个区域，中间有可调节宽度比例的调节手柄
            # 左边为一个组
            VGroup(
                Item("name"),   # 绘图名编辑框
                Item("clear_button"), # 清除按钮
                Heading(u"X轴数据"),  # 静态文本
                # X轴选择器，用EnumEditor编辑器，即ComboBox控件，控件中的
                # data_source的names属性得到
                Item("selected_xaxis", editor=
                    EnumEditor(name="object.data_source.names", for
                Heading(u"Y轴数据"), # 静态文本
                # Y轴选择器，由于Y轴可以多选，因此用CheckBox列表编辑，按两
                Item("selected_items", style="custom",
                     editor=CheckListEditor(name="object.data_sourc
                            cols=2, format_str=u"%s")),
                show_border = True, # 显示组的边框
                scrollable = True,  # 组中的控件过多时，采用滚动条
```

```
                show_labels = False # 组中的所有控件都不显示标签
            ),
            # 右边绘图控件
            Item("figure", editor=MPLFigureEditor(), show_label=Fal
        )
    )

    def _name_changed(self):
        """
当绘图名发生变化时，更新绘图的标题
"""
        axe = self.figure.axes[0]
        axe.set_title(self.name)
        self.figure.canvas.draw()

    def _clear_button_fired(self):
        """
清除按钮的事件处理
"""
        self.selected_items = []
        self.update()

    def _figure_default(self):
        """
figure属性的缺省值，直接创建一个Figure对象
"""
        figure = Figure()
        figure.add_axes([0.05, 0.1, 0.9, 0.85]) #添加绘图区域，四周留有
        return figure

    def _selected_items_changed(self):
        """
Y轴数据选择更新
"""
        self.update()

    def _selected_xaxis_changed(self):
        """
X轴数据选择更新
"""
        self.update()

    def update(self):
        """
重新绘制所有的曲线
"""
        axe = self.figure.axes[0]
        axe.clear()
        try:
            xdata = self.data_source.data[self.selected_xaxis]
        except:
            return
        for field in self.selected_items:
```

```
            axe.plot(xdata, self.data_source.data[field], label=fie
        axe.set_xlabel(self.selected_xaxis)
        axe.set_title(self.name)
        axe.legend()
        self.figure.canvas.draw()

class CSVGrapher(HasTraits):
    """
 主界面包括绘图列表，数据源，文件选择器和添加绘图按钮
 """
    graph_list = List(Instance(Graph)) # 绘图列表
    data_source = Instance(DataSource) # 数据源
    csv_file_name = File(filter=[u"*.csv"]) # 文件选择
    add_graph_button = Button(u"添加绘图") # 添加绘图按钮

    view = View(
        # 整个窗口分为上下两个部分
        VGroup(
            # 上部分横向放置控件，因此用HGroup
            HGroup(
                # 文件选择控件
                Item("csv_file_name", label=u"选择CSV文件", width=40
                # 添加绘图按钮
                Item("add_graph_button", show_label=False)
            ),
            # 下部分是绘图列表，采用ListEditor编辑器显示
            Item("graph_list", style="custom", show_label=False,
                 editor=ListEditor(
                     use_notebook=True, # 是用多标签页格式显示
                     deletable=True, # 可以删除标签页
                     dock_style="tab", # 标签dock样式
                     page_name=".name") # 标题页的文本使用Graph对象的n
                )
        ),
        resizable = True,
        height = 0.8,
        width = 0.8,
        title = u"CSV数据绘图器"
    )

    def _csv_file_name_changed(self):
        """
 打开新文件时的处理，根据文件创建一个DataSource
 """
        self.data_source = DataSource()
        self.data_source.load_csv(self.csv_file_name)
        del self.graph_list[:]

    def _add_graph_button_changed(self):
        """
 添加绘图按钮的事件处理
 """
        if self.data_source != None:
```

```
            self.graph_list.append( Graph(data_source = self.data_s

if __name__ == "__main__":
    csv_grapher = CSVGrapher()
    csv_grapher.configure_traits()
```

NLMS算法的模拟测试

相关文档： 自适应滤波器和*NLMS*模拟

测试NLMS在系统辨识、信号预测和信号均衡方面的应用。

```
# -*- coding: utf-8 -*-
# filename: nlms_test.py

import numpy as np
import pylab as pl
import nlms_numpy
import scipy.signal

# 随机产生FIR滤波器的系数, 长度为length,  延时为delay,  指数衰减
def make_path(delay, length):
   path_length = length - delay
   h = np.zeros(length, np.float64)
   h[delay:] = np.random.standard_normal(path_length) * np.exp( np
   h /= np.sqrt(np.sum(h*h))
   return h

def plot_converge(y, u, label=""):
    size = len(u)
    avg_number = 200
    e = np.power(y[:size] - u, 2)
    tmp = e[:int(size/avg_number)*avg_number]
    tmp.shape = -1, avg_number
    avg = np.average( tmp, axis=1 )
    pl.plot(np.linspace(0, size, len(avg)), 10*np.log10(avg), linew

def diff_db(h0, h):
   return 10*np.log10(np.sum((h0-h)*(h0-h)) / np.sum(h0*h0))

# 用NLMS进行系统辨识的模拟, 未知系统的传递函数为h0, 使用的参照信号为x
def sim_system_identify(nlms, x, h0, step_size, noise_scale):
      y = np.convolve(x, h0)
      d = y + np.random.standard_normal(len(y)) * noise_scale # 添力
      h = np.zeros(len(h0), np.float64) # 自适应滤波器的长度和未知系统长
      u = nlms( x, d, h, step_size )
      return y, u, h

def system_identify_test1():
    h0 = make_path(32, 256) # 随机产生一个未知系统的传递函数
    x = np.random.standard_normal(10000)  # 参照信号为白噪声
    y, u, h = sim_system_identify(nlms_numpy.nlms, x, h0, 0.5, 0.1)
    print diff_db(h0, h)
    pl.figure( figsize=(8, 6) )
    pl.subplot(211)
```

```
    pl.subplots_adjust(hspace=0.4)
    pl.plot(h0, c="r")
    pl.plot(h, c="b")
    pl.title(u"未知系统和收敛后的滤波器的系数比较")
    pl.subplot(212)
    plot_converge(y, u)
    pl.title(u"自适应滤波器收敛特性")
    pl.xlabel("Iterations (samples)")
    pl.ylabel("Converge Level (dB)")
    pl.show()

def system_identify_test2():
    h0 = make_path(32, 256) # 随机产生一个未知系统的传递函数
    x = np.random.standard_normal(20000)  # 参照信号为白噪声
    pl.figure(figsize=(8,4))
    for step_size in np.arange(0.1, 1.0, 0.2):
        y, u, h = sim_system_identify(nlms_numpy.nlms, x, h0, step_
        plot_converge(y, u, label=u"μ=%s" % step_size)
    pl.title(u"更新系数和收敛特性的关系")
    pl.xlabel("Iterations (samples)")
    pl.ylabel("Converge Level (dB)")
    pl.legend()
    pl.show()

def system_identify_test3():
    h0 = make_path(32, 256) # 随机产生一个未知系统的传递函数
    x = np.random.standard_normal(20000)  # 参照信号为白噪声
    pl.figure(figsize=(8,4))
    for noise_scale in [0.05, 0.1, 0.2, 0.4, 0.8]:
        y, u, h = sim_system_identify(nlms_numpy.nlms, x, h0, 0.5,
        plot_converge(y, u, label=u"noise=%s" % noise_scale)
    pl.title(u"外部干扰和收敛特性的关系")
    pl.xlabel("Iterations (samples)")
    pl.ylabel("Converge Level (dB)")
    pl.legend()
    pl.show()

def sim_signal_equation(nlms, x, h0, D, step_size, noise_scale):
    d = x[:-D]
    x = x[D:]
    y = np.convolve(x, h0)[:len(x)]
    h = np.zeros(2*len(h0)+2*D, np.float64)
    y += np.random.standard_normal(len(y)) * noise_scale
    u = nlms(y, d, h, step_size)
    return h

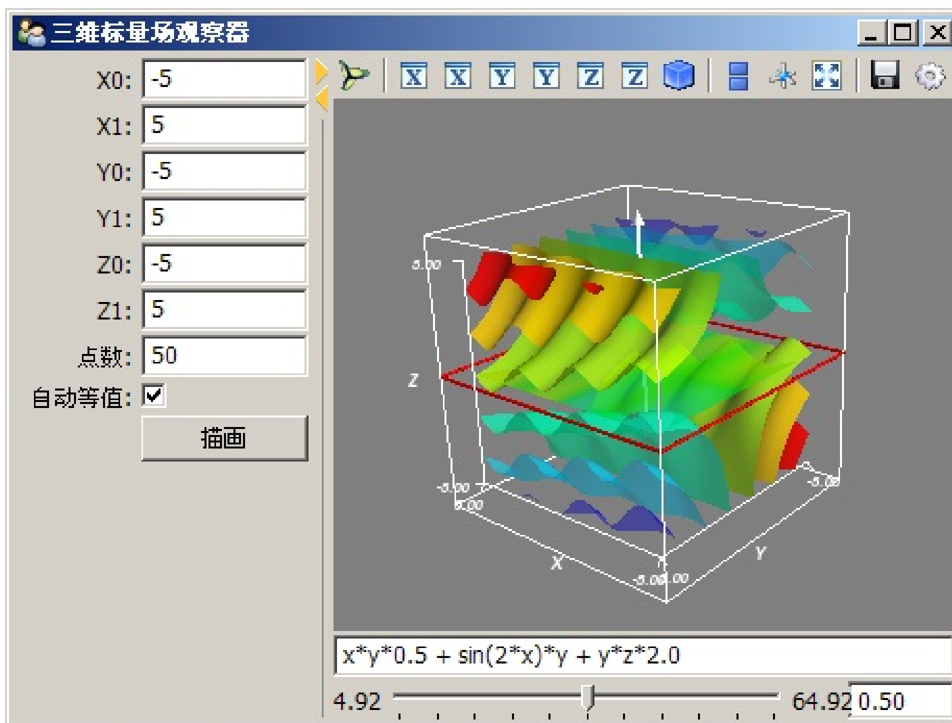
def signal_equation_test1():
    h0 = make_path(5, 64)
    D = 128
    length = 20000
    data = np.random.standard_normal(length+D)
    h = sim_signal_equation(nlms_numpy.nlms, data, h0, D, 0.5, 0.1)
    pl.figure(figsize=(8,4))
```

```
    pl.plot(h0, label=u"未知系统")
    pl.plot(h, label=u"自适应滤波器")
    pl.plot(np.convolve(h0, h), label=u"二者卷积")
    pl.title(u"信号均衡演示")
    pl.legend()
    w0, H0 = scipy.signal.freqz(h0, worN = 1000)
    w, H = scipy.signal.freqz(h, worN = 1000)
    pl.figure(figsize=(8,4))
    pl.plot(w0, 20*np.log10(np.abs(H0)), w, 20*np.log10(np.abs(H))
    pl.title(u"未知系统和自适应滤波器的振幅特性")
    pl.xlabel(u"圆频率")
    pl.ylabel(u"振幅(dB)")
    pl.show()

signal_equation_test1()
```

三维标量场观察器

相关文档： 将*Mayavi*嵌入到界面中




```
# -*- coding: utf-8 -*-
import numpy as np
from numpy import *

from enthought.traits.api import *
from enthought.traits.ui.api import *
from enthought.tvtk.pyface.scene_editor import SceneEditor
from enthought.mayavi.tools.mlab_scene_model import MlabSceneModel
from enthought.mayavi.core.ui.mayavi_scene import MayaviScene

class FieldViewer(HasTraits):
    """三维标量场观察器"""

    # 三个轴的取值范围
    x0, x1 = Float(-5), Float(5)
    y0, y1 = Float(-5), Float(5)
    z0, z1 = Float(-5), Float(5)
    points = Int(50) # 分割点数
    autocontour = Bool(True) # 是否自动计算等值面
    v0, v1 = Float(0.0), Float(1.0) # 等值面的取值范围
    contour = Range("v0", "v1", 0.5) # 等值面的值
    function = Str("x*x*0.5 + y*y + z*z*2.0") # 标量场函数
    plotbutton = Button(u"描画")
    scene = Instance(MlabSceneModel, ()) # mayavi场景
```

```
    view = View(
        HSplit(
            VGroup(
                "x0","x1","y0","y1","z0","z1",
                Item('points', label=u"点数"),
                Item('autocontour', label=u"自动等值"),
                Item('plotbutton', show_label=False),
            ),
            VGroup(
                Item(name='scene',
                    editor=SceneEditor(scene_class=MayaviScene), #
                    resizable=True,
                    height=300,
                    width=350
                ), 'function',
                Item('contour',
                    editor=RangeEditor(format="%1.2f",
                        low_name="v0", high_name="v1")
                ), show_labels=False
            )
        ),
        width = 500, resizable=True, title=u"三维标量场观察器"
    )

    def _plotbutton_fired(self):
        self.plot()

    def _autocontour_changed(self):
        "自动计算等值平面的设置改变事件响应"
        if hasattr(self, "g"):
            self.g.contour.auto_contours = self.autocontour
            if not self.autocontour:
                self._contour_changed()

    def _contour_changed(self):
        "等值平面的值改变事件响应"
        if hasattr(self, "g"):
            if not self.g.contour.auto_contours:
                self.g.contour.contours = [self.contour]

    def plot(self):
        "绘制场景"
        # 产生三维网格
        x, y, z = mgrid[
            self.x0:self.x1:1j*self.points,
            self.y0:self.y1:1j*self.points,
            self.z0:self.z1:1j*self.points]
        scalars = eval(self.function) # 根据函数计算标量场的值
        self.scene.mlab.clf() # 清空当前场景

        # 绘制等值平面
        g = self.scene.mlab.contour3d(x, y, z, scalars, contours=8,
        g.contour.auto_contours = self.autocontour
```

```
        self.scene.mlab.axes() # 添加坐标轴

        # 添加一个X-Y的切面
        s = self.scene.mlab.pipeline.scalar_cut_plane(g)
        cutpoint = (self.x0+self.x1)/2, (self.y0+self.y1)/2, (self.
        s.implicit_plane.normal = (0,0,1) # x cut
        s.implicit_plane.origin = cutpoint

        self.g = g
        self.scalars = scalars
        # 计算标量场的值的范围
        self.v0 = np.min(scalars)
        self.v1 = np.max(scalars)

app = FieldViewer()
app.configure_traits()
```

频谱泄漏和hann窗

相关文档： 频域信号处理

对于8kHz取样频率的200Hz 300Hz的叠加波形进行512点FFT计算其频谱，比较矩形窗和hann窗的频谱泄漏。

```
# -*- coding: utf-8 -*-
#用hann窗降低频谱泄漏
#
import numpy as np
import pylab as pl
import scipy.signal as signal

sampling_rate = 8000
fft_size = 512
t = np.arange(0, 1.0, 1.0/sampling_rate)
x = np.sin(2*np.pi*200*t)  + 2*np.sin(2*np.pi*300*t)

xs = x[:fft_size]
ys = xs * signal.hann(fft_size, sym=0)

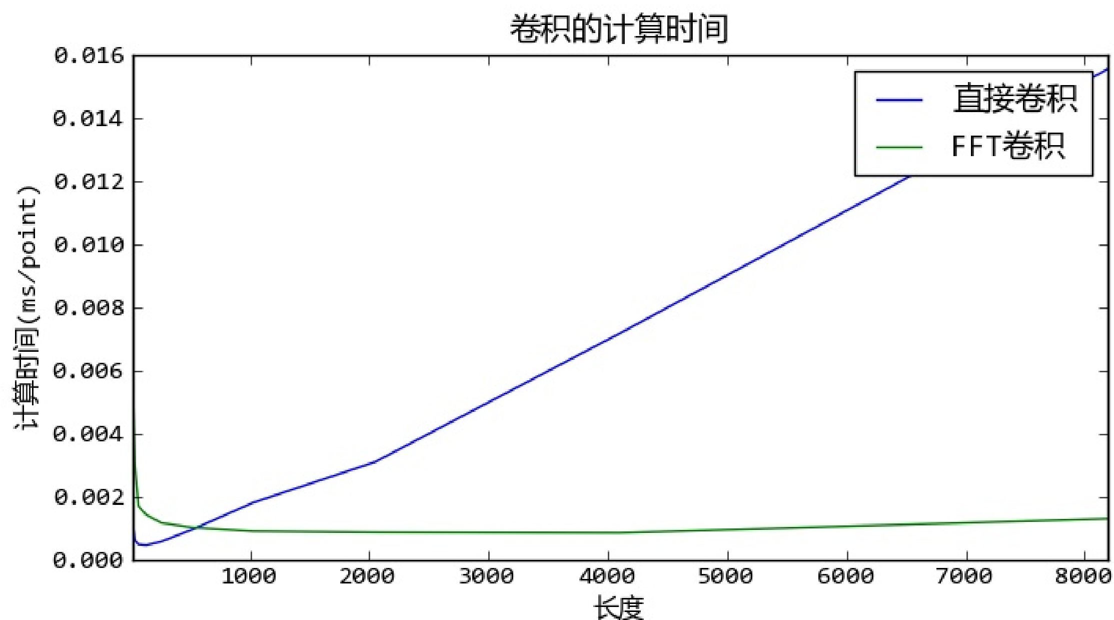
xf = np.fft.rfft(xs)/fft_size
yf = np.fft.rfft(ys)/fft_size
freqs = np.linspace(0, sampling_rate/2, fft_size/2+1)
xfp = 20*np.log10(np.clip(np.abs(xf), 1e-20, 1e100))
yfp = 20*np.log10(np.clip(np.abs(yf), 1e-20, 1e100))
pl.figure(figsize=(8,4))
pl.title(u"200Hz和300Hz的波形和频谱")
pl.plot(freqs, xfp, label=u"矩形窗")
pl.plot(freqs, yfp, label=u"hann窗")
pl.legend()
pl.xlabel(u"频率(Hz)")

a = pl.axes([.4, .2, .4, .4])
a.plot(freqs, xfp, label=u"矩形窗")
a.plot(freqs, yfp, label=u"hann窗")
a.set_xlim(100, 400)
a.set_ylim(-40, 0)
pl.show()
```

FFT卷积的速度比较

相关文档： 频域信号处理

直接卷积的复杂度为O(NN)，*FFT*的复杂度为*O(N*log(N))，此程序分别计算直接卷积和快速卷积的耗时曲线。请注意Y轴为每点的平均运算时间。



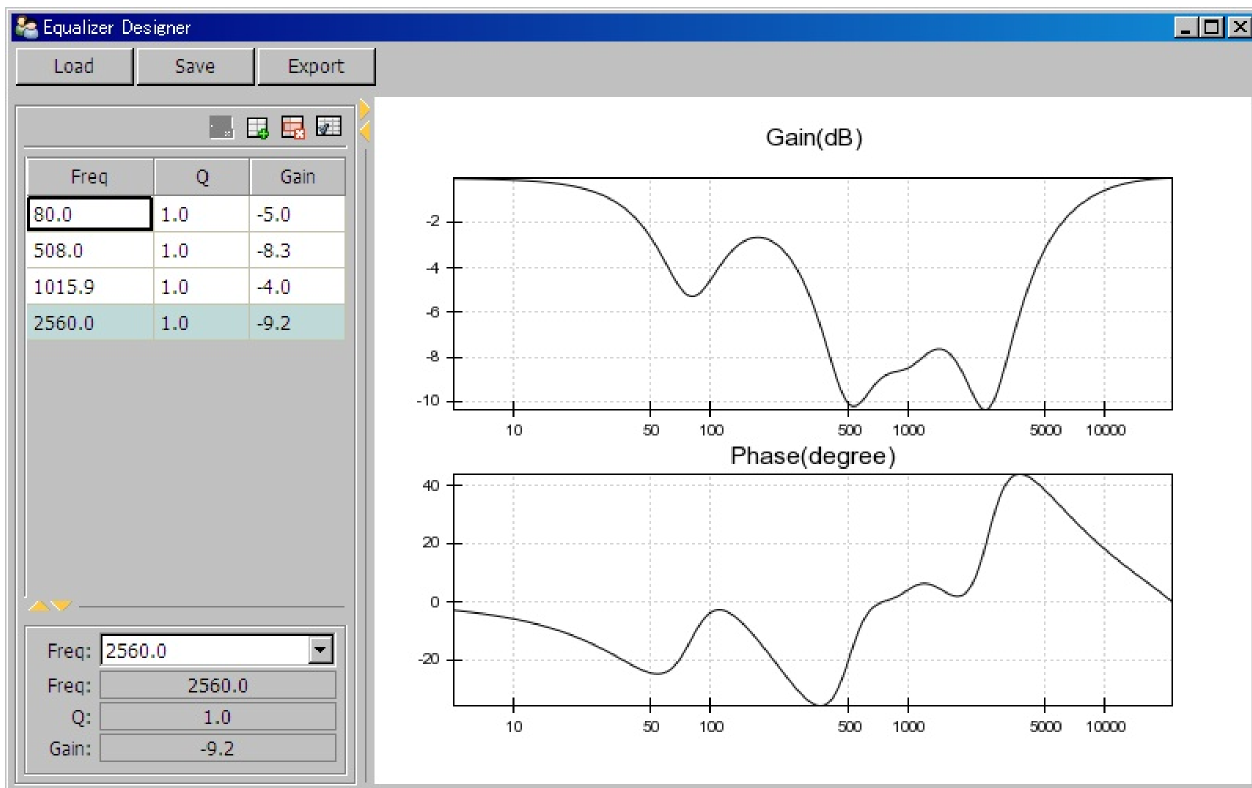
```
# -*- coding: utf-8 -*-
import numpy as np
import timeit
def fft_convolve(a,b):
    n = len(a)+len(b)-1
    N = 2**(int(np.log2(n))+1)
    A = np.fft.fft(a, N)
    B = np.fft.fft(b, N)
    return np.fft.ifft(A*B)[:n]

if __name__ == "__main__":
    from pylab import *
    n_list = []
    t1_list = []
    t2_list = []
    for n in xrange(4, 14):
        N = 2**n
        count = 10000**2 / N**2
        if count > 10000: count = 10000
        setup = """
import numpy as np
from __main__ import fft_convolve
a = np.random.rand(%s)
b = np.random.rand(%s)
 """ % (N, N)
        t1 = timeit.timeit("np.convolve(a,b)", setup, number=count
        t2 = timeit.timeit("fft_convolve(a,b)", setup, number=count
        t1_list.append(t1*1000/count/N)
        t2_list.append(t2*1000/count/N)
        n_list.append(N)
    figure(figsize=(8,4))
    plot(n_list, t1_list, label=u"直接卷积")
    plot(n_list, t2_list, label=u"FFT卷积")
    legend()
    title(u"卷积的计算时间")
    ylabel(u"计算时间(ms/point)")
    xlabel(u"长度")
    xlim(min(n_list),max(n_list))
    show()
```

二次均衡器设计

相关文档： 数字信号系统

用Traits.UI和Chaco制作的二次均衡器的设计工具，用户可以任意添加二次滤波器，并且调整其中心频率、增益和Q值，并即时查看组合之后的最终频率响应。




```
# -*- coding: utf-8 -*-
import math
from enthought.traits.api import Float, HasTraits, List, Array, on_
from enthought.traits.ui.api import View, TableEditor, Item, Group,
from enthought.traits.ui.table_column import ObjectColumn
from enthought.chaco.api import Plot, AbstractPlotData, ArrayPlotDa
from enthought.chaco.tools.api import PanTool, ZoomTool
from enthought.enable.api import Component, ComponentEditor
from enthought.pyface.api import FileDialog, OK
import pickle
import numpy as np

SAMPLING_RATE = 44100.0 # 取样频率
WORN = 1000 # 频率响应曲线的点数

# 对数圆频率数组
W = np.logspace(np.log10(10/SAMPLING_RATE*np.pi), np.log10(np.pi),

# 对数频率数组
FREQS = W / 2 / np.pi * SAMPLING_RATE
```

```
# 候选频率
EQ_FREQS = [20.0,25.2,31.7,40.0,50.4,63.5,80.0,100.8,
    127.0,160.0,201.6,254.0,320.0,403.2,508.0,640.0,
    806.3,1015.9,1280.0,1612.7,2031.9,2560.0,3225.4,
    4063.7,    5120.0,    6450.8,    8127.5,    10240.0,12901.6,
    16255.0,20480.0,]

def scrubber(inc):
    '''创建不同增量的ScrubberEditor'''
    return ScrubberEditor(
        hover_color  = 0xFFFFFF,
        active_color = 0xA0CD9E,
        border_color = 0x808080,
        increment = inc
    )

def myfreqz(b, a, w):
    '''计算滤波器在w个点的频率响应'''
    zm1 = np.exp(-1j*w)
    h = np.polyval(b[::-1], zm1) / np.polyval(a[::-1], zm1)
    return h

def design_equalizer(freq, Q, gain, Fs):
    '''设计二次均衡滤波器的系数'''
    A = 10**(gain/40.0)
    w0 = 2*math.pi*freq/Fs
    alpha = math.sin(w0) / 2 / Q

    b0 = 1 + alpha * A
    b1 = -2*math.cos(w0)
    b2 = 1 - alpha * A
    a0 = 1 + alpha / A
    a1 = -2*math.cos(w0)
    a2 = 1 - alpha / A

    return [b0/a0,b1/a0,b2/a0], [1.0, a1/a0, a2/a0]

class Equalizer(HasTraits):
    freq = Range(10.0, SAMPLING_RATE/2, 1000)
    Q = Range(0.1, 10.0, 1.0)
    gain = Range(-24.0, 24.0, 0)

    a = List(Float, [1.0,0.0,0.0])
    b = List(Float, [1.0,0.0,0.0])

    h = Array(dtype=np.complex, transient = True)

    def __init__(self):
        super(Equalizer, self).__init__()
        self.design_parameter()

    @on_trait_change("freq,Q,gain")
```

```
    def design_parameter(self):
        '''设计系数并计算频率响应'''
        try:
            self.b, self.a = design_equalizer(self.freq, self.Q, se
        except:
            self.b, self.a = [1.0,0.0,0.0], [1.0,0.0,0.0]
        self.h = myfreqz(self.b, self.a, W)

    def export_parameters(self, f):
        '''输出滤波器系数为C语言数组'''
        tmp = self.b[0], self.b[1], self.b[2], self.a[1], self.a[2
        f.write("{%s,%s,%s,%s,%s}, // %s,%s,%s\n" % tmp)

class Equalizers(HasTraits):
    eqs = List(Equalizer, [Equalizer()])
    h = Array(dtype=np.complex, transient = True)

    # Equalizer列表eqs的编辑器定义
    table_editor = TableEditor(
        columns = [
            ObjectColumn(name="freq", width=0.4, style="readonly"),
            ObjectColumn(name="Q", width=0.3, style="readonly"),
            ObjectColumn(name="gain", width=0.3, style="readonly"),
        ],
        deletable = True,
        sortable = True,
        auto_size = False,
        show_toolbar = True,
        edit_on_first_click = False,
        orientation = 'vertical',
        edit_view = View(
            Group(
                Item("freq", editor=EnumEditor(values=EQ_FREQS)),
                Item("freq", editor=scrubber(1.0)),
                Item("Q", editor=scrubber(0.01)),
                Item("gain", editor=scrubber(0.1)),
                show_border=True,
            ),
            resizable = True
        ),
        row_factory = Equalizer
    )

    view = View(
        Item("eqs", show_label=False, editor=table_editor),
        width     = 0.25,
        height    = 0.5,
        resizable = True
    )

    @on_trait_change("eqs.h")
    def recalculate_h(self):
        '''计算多组均衡器级联时的频率响应'''
```

```
        try:
            tmp = np.array([eq.h for eq in self.eqs if eq.h != None
            self.h = np.prod(tmp, axis=0)
        except:
            pass

    def export(self, path):
        '''将均衡器的系数输出为C语言文件'''
        f = file(path, "w")
        f.write("double EQ_PARS[][5] = {\n")
        f.write("//b0,b1,b2,a0,a1 // frequency, Q, gain\n")
        for eq in self.eqs:
            eq.export_parameters(f)
        f.write("};\n")
        f.close()

class EqualizerDesigner(HasTraits):
    '''均衡器设计器的主界面'''

    equalizers = Instance(Equalizers)

    # 保存绘图数据的对象
    plot_data = Instance(AbstractPlotData)

    # 绘制波形图的容器
    container = Instance(Component)

    plot_gain = Instance(Component)
    plot_phase = Instance(Component)
    save_button = Button("Save")
    load_button = Button("Load")
    export_button = Button("Export")

    view = View(
        VGroup(
            HGroup(
                Item("load_button"),
                Item("save_button"),
                Item("export_button"),
                show_labels = False
            ),
            HSplit(
                VGroup(
                    Item("equalizers", style="custom", show_label=F
                    show_border=True,
                ),
                Item("container", editor=ComponentEditor(size=(800,
            )
        ),
        resizable = True,
        width = 800,
        height = 500,
        title = u"Equalizer Designer"
```

```
    )

    def _create_plot(self, data, name, type="line"):
        p = Plot(self.plot_data)
        p.plot(data, name=name, title=name, type=type)
        p.tools.append(PanTool(p))
        zoom = ZoomTool(component=p, tool_mode="box", always_on=Fal
        p.overlays.append(zoom)
        p.title = name
        p.index_scale = "log"
        return p

    def __init__(self):
        super(EqualizerDesigner, self).__init__()
        self.plot_data = ArrayPlotData(f=FREQS, gain=[], phase=[])
        self.plot_gain = self._create_plot(("f", "gain"), "Gain(dB
        self.plot_phase = self._create_plot(("f", "phase"), "Phase(
        self.container = VPlotContainer()
        self.container.add( self.plot_phase )
        self.container.add( self.plot_gain )
        self.plot_gain.padding_bottom = 20
        self.plot_phase.padding_top = 20

    def _equalizers_default(self):
        return Equalizers()

    @on_trait_change("equalizers.h")
    def redraw(self):
        gain = 20*np.log10(np.abs(self.equalizers.h))
        phase = np.angle(self.equalizers.h, deg=1)
        self.plot_data.set_data("gain", gain)
        self.plot_data.set_data("phase", phase)

    def _save_button_fired(self):
        dialog = FileDialog(action="save as", wildcard='EQ files (
        result = dialog.open()
        if result == OK:
            f = file(dialog.path, "wb")
            pickle.dump( self.equalizers , f)
            f.close()

    def _load_button_fired(self):
        dialog = FileDialog(action="open", wildcard='EQ files (*.eq
        result = dialog.open()
        if result == OK:
            f = file(dialog.path, "rb")
            self.equalizers = pickle.load(f)
            f.close()

    def _export_button_fired(self):
        dialog = FileDialog(action="save as", wildcard='c files (*.
        result = dialog.open()
        if result == OK:
```

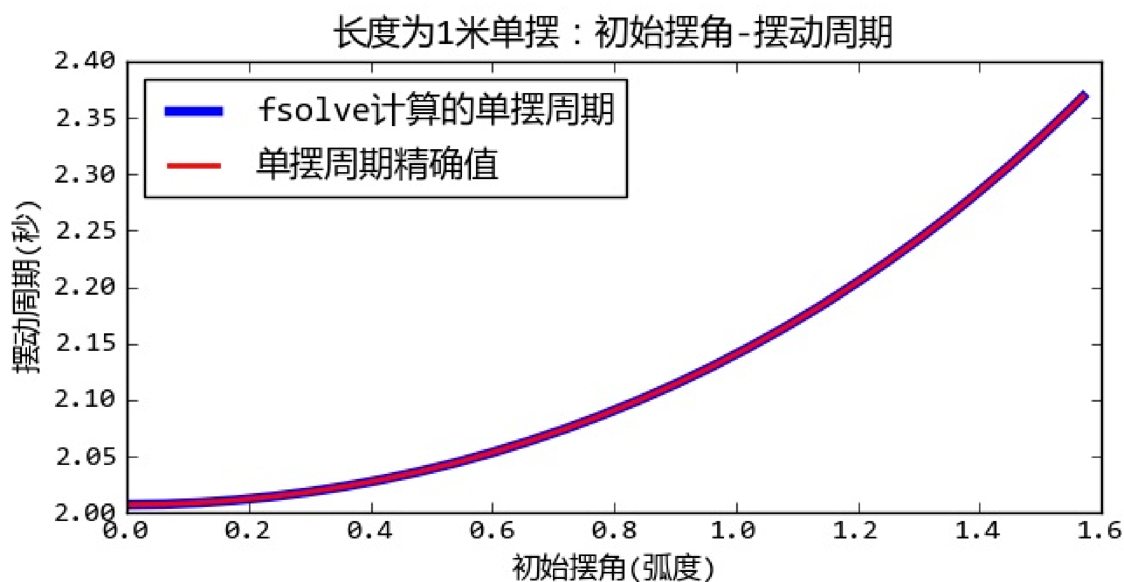
```
            self.equalizers.export(dialog.path)

win = EqualizerDesigner()
win.configure_traits()
```

单摆摆动周期的计算

相关文档： 单摆和双摆模拟

本程序利用odeint和fsolve计算单摆的摆动周期，并且和精确值进行比较。



```
# -*- coding: utf-8 -*-
from math import sin, sqrt
import numpy as np
from scipy.integrate import odeint
from scipy.optimize import fsolve
import pylab as pl
from scipy.special import ellipk

g = 9.8

def pendulum_equations(w, t, l):
    th, v = w
    dth = v
    dv  = - g/l * sin(th)
    return dth, dv

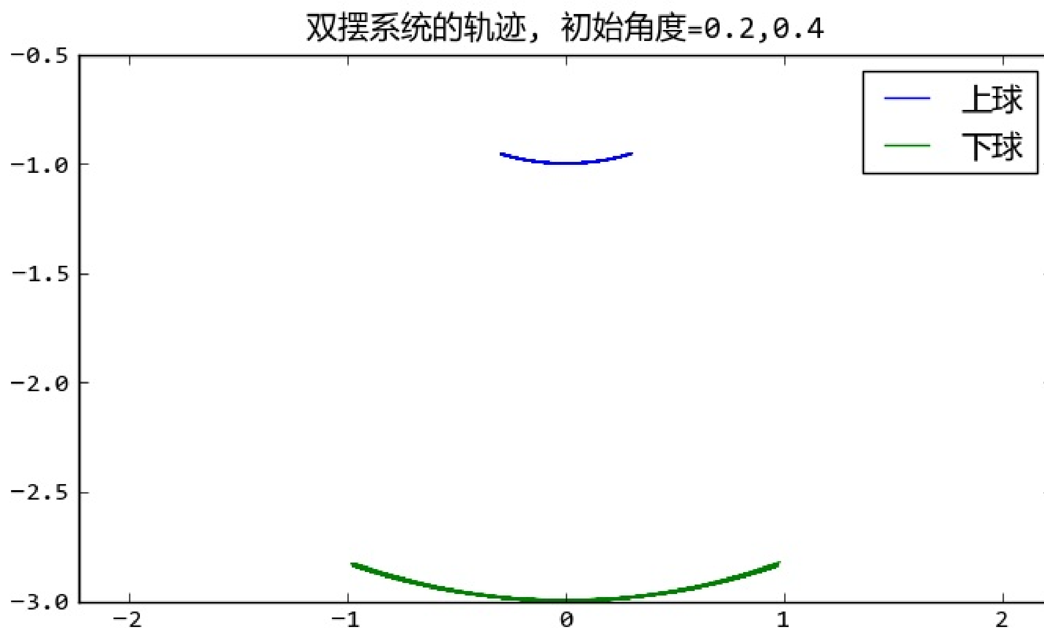
def pendulum_th(t, l, th0):
    track = odeint(pendulum_equations, (th0, 0), [0, t], args=(l,))
    return track[-1, 0]

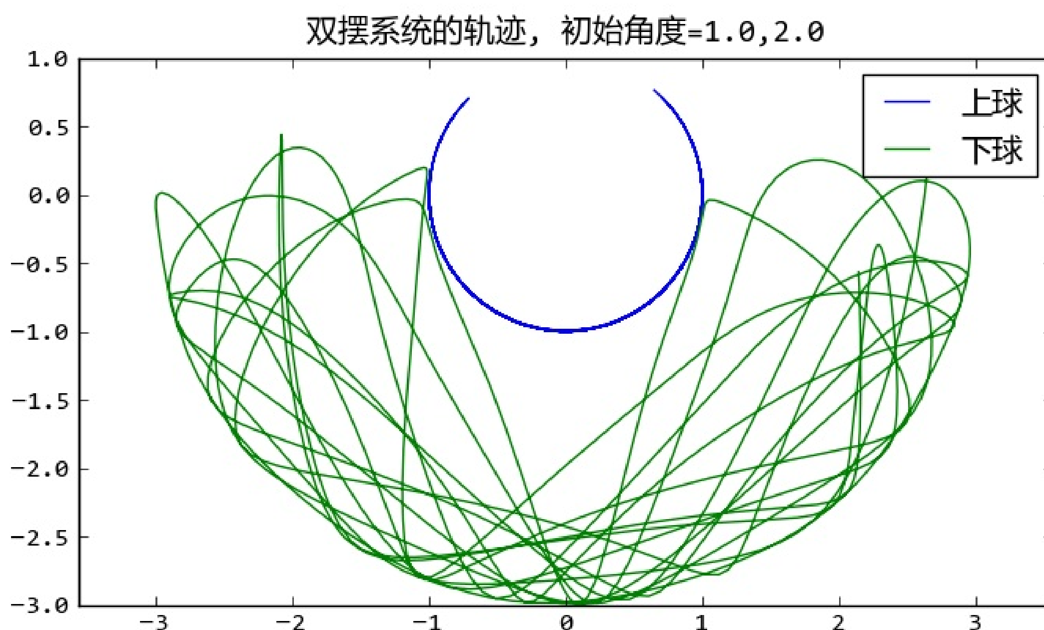
def pendulum_period(l, th0):
    t0 = 2*np.pi*sqrt( l/g ) / 4
    t = fsolve( pendulum_th, t0, args = (l, th0) )
    return t*4

ths = np.arange(0, np.pi/2.0, 0.01)
periods = [pendulum_period(1, th) for th in ths]
periods2 = 4*sqrt(1.0/g)*ellipk(np.sin(ths/2)**2) # 计算单摆周期的精确
pl.plot(ths, periods, label = u"fsolve计算的单摆周期", linewidth=4.0)
pl.plot(ths, periods2, "r", label = u"单摆周期精确值", linewidth=2.0)
pl.legend(loc='upper left')
pl.title(u"长度为1米单摆：初始摆角-摆动周期")
pl.xlabel(u"初始摆角(弧度)")
pl.ylabel(u"摆动周期(秒)")
pl.show()
```

双摆系统的动画模拟

相关文档： 单摆和双摆模拟





用odeint解双摆系统

文件名: double_pendulum_odeint.py

```
# -*- coding: utf-8 -*-
```

```
from math import sin,cos
import numpy as np
from scipy.integrate import odeint

g = 9.8

class DoublePendulum(object):
    def __init__(self, m1, m2, l1, l2):
        self.m1, self.m2, self.l1, self.l2 = m1, m2, l1, l2
        self.init_status = np.array([0.0,0.0,0.0,0.0])

    def equations(self, w, t):
        """
 微分方程公式
 """
        m1, m2, l1, l2 = self.m1, self.m2, self.l1, self.l2
        th1, th2, v1, v2 = w
        dth1 = v1
        dth2 = v2

        #eq of th1
        a = l1*l1*(m1+m2)  # dv1 parameter
        b = l1*m2*l2*cos(th1-th2) # dv2 paramter
        c = l1*(m2*l2*sin(th1-th2)*dth2*dth2 + (m1+m2)*g*sin(th1))

        #eq of th2
        d = m2*l2*l1*cos(th1-th2) # dv1 parameter
        e = m2*l2*l2 # dv2 parameter
        f = m2*l2*(-l1*sin(th1-th2)*dth1*dth1 + g*sin(th2))

        dv1, dv2 = np.linalg.solve([[a,b],[d,e]], [-c,-f])

        return np.array([dth1, dth2, dv1, dv2])

def double_pendulum_odeint(pendulum, ts, te, tstep):
    """
 对双摆系统的微分方程组进行数值求解，返回两个小球的X-Y坐标
 """
    t = np.arange(ts, te, tstep)
    track = odeint(pendulum.equations, pendulum.init_status, t)
    th1_array, th2_array = track[:,0], track[:, 1]
    l1, l2 = pendulum.l1, pendulum.l2
    x1 = l1*np.sin(th1_array)
    y1 = -l1*np.cos(th1_array)
    x2 = x1 + l2*np.sin(th2_array)
    y2 = y1 - l2*np.cos(th2_array)
    pendulum.init_status = track[-1,:].copy() #将最后的状态赋给pendul
    return [x1, y1, x2, y2]

if __name__ == "__main__":
    import matplotlib.pyplot as pl
    pendulum = DoublePendulum(1.0, 2.0, 1.0, 2.0)
    th1, th2 = 1.0, 2.0
```

```
    pendulum.init_status[:2] = th1, th2
    x1, y1, x2, y2 = double_pendulum_odeint(pendulum, 0, 30, 0.02)
    pl.plot(x1,y1, label = u"上球")
    pl.plot(x2,y2, label = u"下球")
    pl.title(u"双摆系统的轨迹, 初始角度=%s,%s" % (th1, th2))
    pl.legend()
    pl.axis("equal")
    pl.show()
```

摆动动画

文件名: double_pendulum_animation.py

```
# -*- coding: utf-8 -*-
import matplotlib
matplotlib.use('WXAgg') # do this before importing pylab
import matplotlib.pyplot as pl
from double_pendulum_odeint import double_pendulum_odeint, DoublePe

fig = pl.figure(figsize=(4,4))
line1, = pl.plot([0,0], [0,0], "-o")
line2, = pl.plot([0,0], [0,0], "-o")
pl.axis("equal")
pl.xlim(-4,4)
pl.ylim(-4,2)

pendulum = DoublePendulum(1.0, 2.0, 1.0, 2.0)
pendulum.init_status[:] = 1.0, 2.0, 0, 0

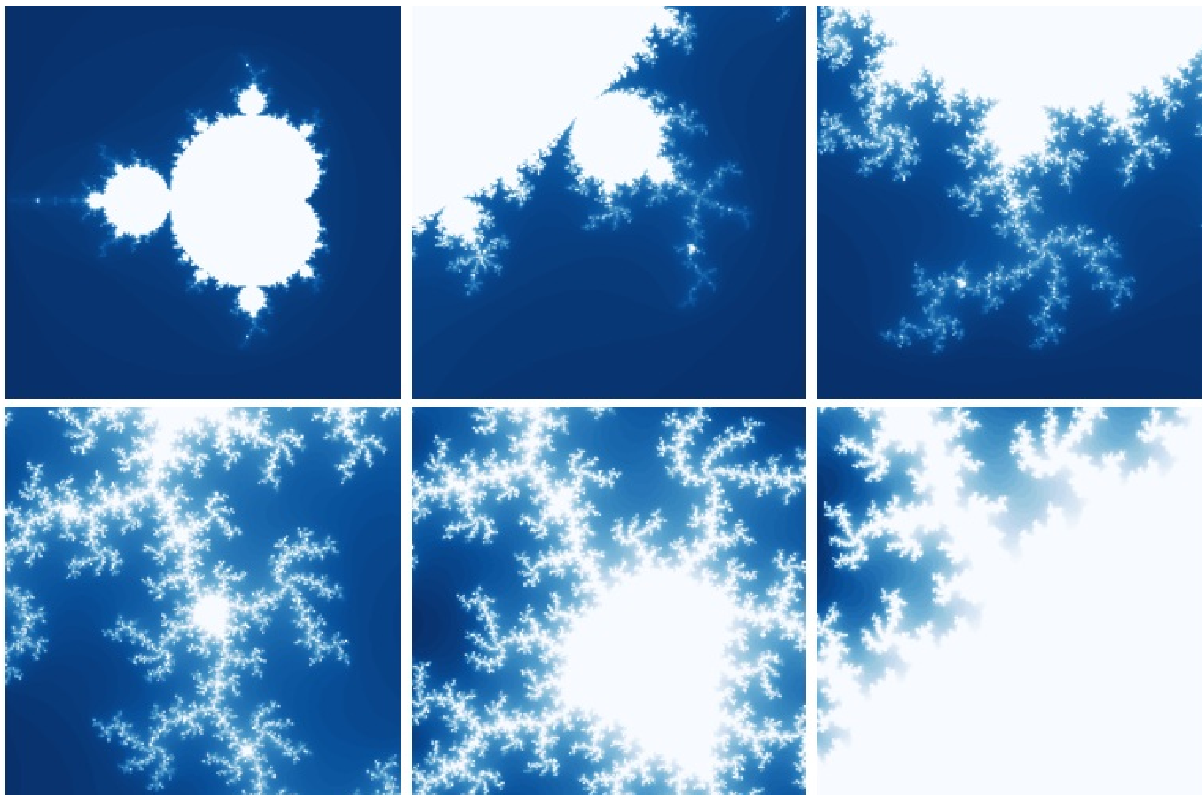
x1, y1, x2, y2 = [],[],[],[]
idx = 0

def update_line(event):
    global x1, x2, y1, y2, idx
    if idx == len(x1):
        x1, y1, x2, y2 = double_pendulum_odeint(pendulum, 0, 1, 0.0
        idx = 0
    line1.set_xdata([0, x1[idx]])
    line1.set_ydata([0, y1[idx]])
    line2.set_xdata([x1[idx], x2[idx]])
    line2.set_ydata([y1[idx], y2[idx]])
    fig.canvas.draw()
    idx += 1

import wx
id = wx.NewId()
actor = fig.canvas.manager.frame
timer = wx.Timer(actor, id=id)
timer.Start(1)
wx.EVT_TIMER(actor, id, update_line)
pl.show()
```

绘制Mandelbrot集合

相关文档： *Mandelbrot集合*



纯Python计算版本

```
# -*- coding: utf-8 -*-

import numpy as np
import pylab as pl
import time
from matplotlib import cm

def iter_point(c):
    z = c
    for i in xrange(1, 100): # 最多迭代100次
        if abs(z)>2: break # 半径大于2则认为逃逸
        z = z*z+c
    return i # 返回迭代次数

def draw_mandelbrot(cx, cy, d):
    """
 绘制点(cx, cy)附近正负d的范围的Mandelbrot
 """
    x0, x1, y0, y1 = cx-d, cx+d, cy-d, cy+d
    y, x = np.ogrid[y0:y1:200j, x0:x1:200j]
    c = x + y*1j
    start = time.clock()
    mandelbrot = np.frompyfunc(iter_point,1,1)(c).astype(np.float)
    print "time=",time.clock() - start
    pl.imshow(mandelbrot, cmap=cm.Blues_r, extent=[x0,x1,y0,y1])
    pl.gca().set_axis_off()

x,y = 0.27322626, 0.595153338

pl.subplot(231)
draw_mandelbrot(-0.5,0,1.5)
for i in range(2,7):
    pl.subplot(230+i)
    draw_mandelbrot(x, y, 0.2**(i-1))
pl.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0)
pl.show()
```

Weave版本

```
# -*- coding: utf-8 -*-

import numpy as np
import pylab as pl
import time
import scipy.weave as weave
from matplotlib import cm

def weave_iter_point(c):
    code = """
 std::complex<double> z;
 int i;
 z = c;
 for(i=1;i<100;i++)
 {
 if(std::abs(z) > 2) break;
 z = z*z+c;
 }
 return_val=i;
 """

    f = weave.inline(code, ["c"], compiler="gcc")
    return f

def draw_mandelbrot(cx, cy, d,N=200):
    """
 绘制点(cx, cy)附近正负d的范围的Mandelbrot
 """
    x0, x1, y0, y1 = cx-d, cx+d, cy-d, cy+d
    y, x = np.ogrid[y0:y1:N*1j, x0:x1:N*1j]
    c = x + y*1j
    start = time.clock()
    mandelbrot = np.frompyfunc(weave_iter_point,1,1)(c).astype(np.f
    print "time=",time.clock() - start
    pl.imshow(mandelbrot, cmap=cm.Blues_r, extent=[x0,x1,y0,y1])
    pl.gca().set_axis_off()

x,y = 0.27322626, 0.595153338

pl.subplot(231)
draw_mandelbrot(-0.5,0,1.5)
for i in range(2,7):
    pl.subplot(230+i)
    draw_mandelbrot(x, y, 0.2**(i-1))
pl.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0.02)

pl.show()
```

NumPy加速版本

```
# -*- coding: utf-8 -*-

import numpy as np
import pylab as pl
import time
from matplotlib import cm

def draw_mandelbrot(cx, cy, d, N=200):
    """
 绘制点(cx, cy)附近正负d的范围的Mandelbrot
 """
    global mandelbrot

    x0, x1, y0, y1 = cx-d, cx+d, cy-d, cy+d
    y, x = np.ogrid[y0:y1:N*1j, x0:x1:N*1j]
    c = x + y*1j

    # 创建X,Y轴的坐标数组
    ix, iy = np.mgrid[0:N,0:N]

    # 创建保存mandelbrot图的二维数组，缺省值为最大迭代次数
    mandelbrot = np.ones(c.shape, dtype=np.int)*100

    # 将数组都变成一维的
    ix.shape = -1
    iy.shape = -1
    c.shape = -1
    z = c.copy() # 从c开始迭代，因此开始的迭代次数为1

    start = time.clock()

    for i in xrange(1,100):
        # 进行一次迭代
        z *= z
        z += c
        # 找到所有结果逃逸了的点
        tmp = np.abs(z) > 2.0
        # 将这些逃逸点的迭代次数赋值给mandelbrot图
        mandelbrot[ix[tmp], iy[tmp]] = i

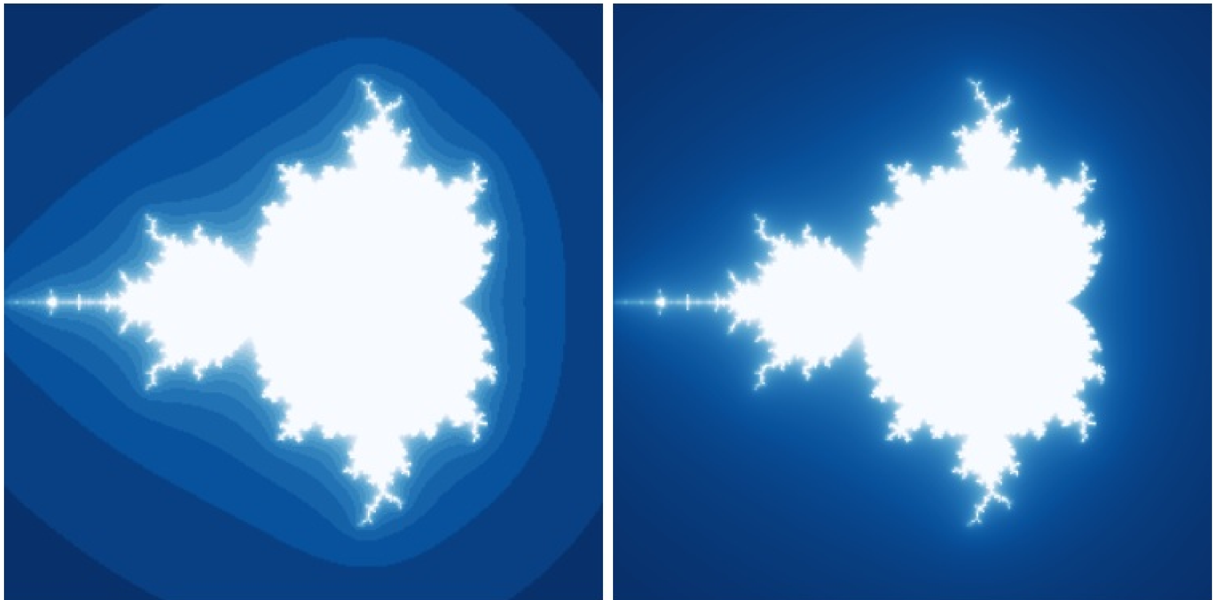
        # 找到所有没有逃逸的点
        np.logical_not(tmp, tmp)
        # 更新ix, iy, c, z只包含没有逃逸的点
        ix,iy,c,z = ix[tmp], iy[tmp], c[tmp],z[tmp]
        if len(z) == 0: break

    print "time=",time.clock() - start
    pl.imshow(mandelbrot, cmap=cm.Blues_r, extent=[x0,x1,y0,y1])
    pl.gca().set_axis_off()

x,y = 0.27322626, 0.595153338
```

```
pl.subplot(231)
draw_mandelbrot(-0.5,0,1.5)
for i in range(2,7):
    pl.subplot(230+i)
    draw_mandelbrot(x, y, 0.2**(i-1))
pl.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0)
pl.show()
```

平滑版本



```
# -*- coding: utf-8 -*-

import numpy as np
import pylab as pl
from math import log
from matplotlib import cm

escape_radius = 10
iter_num = 20

def smooth_iter_point(c):
    z = c
    for i in xrange(1, iter_num):
        if abs(z)>escape_radius: break
        z = z*z+c
    absz = abs(z)
    if absz > 2.0:
        mu = i - log(log(abs(z),2),2)
    else:
        mu = i
    return mu # 返回正规化的迭代次数

def iter_point(c):
    z = c
    for i in xrange(1, iter_num):
        if abs(z)>escape_radius: break
        z = z*z+c
    return i

def draw_mandelbrot(cx, cy, d, N=200):
    global mandelbrot
    """
 绘制点(cx, cy)附近正负d的范围的Mandelbrot
 """
    x0, x1, y0, y1 = cx-d, cx+d, cy-d, cy+d
    y, x = np.ogrid[y0:y1:N*1j, x0:x1:N*1j]
    c = x + y*1j
    mand = np.frompyfunc(iter_point,1,1)(c).astype(np.float)
    smooth_mand = np.frompyfunc(smooth_iter_point,1,1)(c).astype(np
    pl.subplot(121)
    pl.gca().set_axis_off()
    pl.imshow(mand, cmap=cm.Blues_r, extent=[x0,x1,y0,y1])
    pl.subplot(122)
    pl.imshow(smooth_mand, cmap=cm.Blues_r, extent=[x0,x1,y0,y1])
    pl.gca().set_axis_off()

draw_mandelbrot(-0.5,0,1.5,300)
pl.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0)
pl.show()
```

迭代函数系统的分形

相关文档： 迭代函数系统(IFS)



```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as pl
import time

# 蕨类植物叶子的迭代函数和其概率值
eq1 = np.array([[0,0,0],[0,0.16,0]])
p1 = 0.01

eq2 = np.array([[0.2,-0.26,0],[0.23,0.22,1.6]])
p2 = 0.07

eq3 = np.array([[-0.15, 0.28, 0],[0.26,0.24,0.44]])
```

```
p3 = 0.07

eq4 = np.array([[0.85, 0.04, 0],[-0.04, 0.85, 1.6]])
p4 = 0.85

def ifs(p, eq, init, n):
    """
 进行函数迭代
 p: 每个函数的选择概率列表
 eq: 迭代函数列表
 init: 迭代初始点
 n: 迭代次数

 返回值： 每次迭代所得的X坐标数组， Y坐标数组， 计算所用的函数下标
 """

    # 迭代向量的初始化
    pos = np.ones(3, dtype=np.float)
    pos[:2] = init

    # 通过函数概率，计算函数的选择序列
    p = np.add.accumulate(p)
    rands = np.random.rand(n)
    select = np.ones(n, dtype=np.int)*(n-1)
    for i, x in enumerate(p[::-1]):
        select[rands<x] = len(p)-i-1

    # 结果的初始化
    result = np.zeros((n,2), dtype=np.float)
    c = np.zeros(n, dtype=np.float)

    for i in xrange(n):
        eqidx = select[i] # 所选的函数下标
        tmp = np.dot(eq[eqidx], pos) # 进行迭代
        pos[:2] = tmp # 更新迭代向量

        # 保存结果
        result[i] = tmp
        c[i] = eqidx

    return result[:,0], result[:, 1], c

start = time.clock()
x, y, c = ifs([p1,p2,p3,p4],[eq1,eq2,eq3,eq4], [0,0], 100000)
print time.clock() - start
pl.figure(figsize=(6,6))
pl.subplot(121)
pl.scatter(x, y, s=1, c="g", marker="s", linewidths=0)
pl.axis("equal")
pl.axis("off")
pl.subplot(122)
pl.scatter(x, y, s=1,c = c, marker="s", linewidths=0)
pl.axis("equal")
```

```
pl.axis("off")
pl.subplots_adjust(left=0,right=1,bottom=0,top=1,wspace=0,hspace=0)
pl.gcf().patch.set_facecolor("white")
pl.show()
```

迭代函数系统设计器



```
# -*- coding: utf-8 -*-
from enthought.traits.ui.api import *
from enthought.traits.ui.menu import OKCancelButtons
from enthought.traits.api import *
from enthought.traits.ui.wx.editor import Editor

import matplotlib
# matplotlib采用WXAgg为后台，这样才能将绘图控件嵌入以wx为后台界面库的traits
matplotlib.use("WXAgg")
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as
from matplotlib.figure import Figure

import numpy as np
import thread
import time
import wx
import pickle

ITER_COUNT = 4000 # 一次ifs迭代的点数
ITER_TIMES = 10   # 总共调用ifs的次数

def triangle_area(triangle):
```

```
    """
 计算三角形的面积
 """
    A = triangle[0]
    B = triangle[1]
    C = triangle[2]
    AB = A-B
    AC = A-C
    return np.abs(np.cross(AB,AC))/2.0

def solve_eq(triangle1, triangle2):
    """
 解方程，从triangle1变换到triangle2的变换系数
 triangle1,2是二维数组：
 x0,y0
 x1,y1
 x2,y2
 """
    x0,y0 = triangle1[0]
    x1,y1 = triangle1[1]
    x2,y2 = triangle1[2]

    a = np.zeros((6,6), dtype=np.float)
    b = triangle2.reshape(-1)
    a[0, 0:3] = x0,y0,1
    a[1, 3:6] = x0,y0,1
    a[2, 0:3] = x1,y1,1
    a[3, 3:6] = x1,y1,1
    a[4, 0:3] = x2,y2,1
    a[5, 3:6] = x2,y2,1

    c = np.linalg.solve(a, b)
    c.shape = (2,3)
    return c

def ifs(p, eq, init, n):
    """
 进行函数迭代
 p: 每个函数的选择概率列表
 eq: 迭代函数列表
 init: 迭代初始点
 n: 迭代次数

 返回值： 每次迭代所得的X坐标数组， Y坐标数组， 计算所用的函数下标
 """

    # 迭代向量的初始化
    pos = np.ones(3, dtype=np.float)
    pos[:2] = init

    # 通过函数概率，计算函数的选择序列
    p = np.add.accumulate(p)
    rands = np.random.rand(n)
```

```
    select = np.ones(n, dtype=np.int)*(n-1)
    for i, x in enumerate(p[::-1]):
        select[rands<x] = len(p)-i-1

    # 结果的初始化
    result = np.zeros((n,2), dtype=np.float)
    c = np.zeros(n, dtype=np.float)

    for i in xrange(n):
        eqidx = select[i] # 所选的函数下标
        tmp = np.dot(eq[eqidx], pos) # 进行迭代
        pos[:2] = tmp # 更新迭代向量

        # 保存结果
        result[i] = tmp
        c[i] = eqidx

    return result[:,0], result[:, 1], c

class _MPLFigureEditor(Editor):
    """
 使用matplotlib figure的traits编辑器
 """
    scrollable = True

    def init(self, parent):
        self.control = self._create_canvas(parent)

    def update_editor(self):
        pass

    def _create_canvas(self, parent):
        panel = wx.Panel(parent, -1, style=wx.CLIP_CHILDREN)
        sizer = wx.BoxSizer(wx.VERTICAL)
        panel.SetSizer(sizer)
        mpl_control = FigureCanvas(panel, -1, self.value)
        sizer.Add(mpl_control, 1, wx.LEFT | wx.TOP | wx.GROW)
        self.value.canvas.SetMinSize((10,10))
        return panel

class MPLFigureEditor(BasicEditorFactory):
    """
 相当于traits.ui中的EditorFactory，它返回真正创建控件的类
 """
    klass = _MPLFigureEditor

class IFSTriangles(HasTraits):
    """
 三角形编辑器
 """
    version = Int(0) # 三角形更新标志

    def __init__(self, ax):
```

```
        super(IFSTriangles, self).__init__()
        self.colors = ["r","g","b","c","m","y","k"]
        self.points = np.array([(0,0),(2,0),(2,4),(0,1),(1,1),(1,3
        self.equations = self.get_eqs()
        self.ax = ax
        self.ax.set_ylim(-10,10)
        self.ax.set_xlim(-10,10)
        canvas = ax.figure.canvas
        # 绑定canvas的鼠标事件
        canvas.mpl_connect('button_press_event', self.button_press_
        canvas.mpl_connect('button_release_event', self.button_rele
        canvas.mpl_connect('motion_notify_event', self.motion_noti
        self.canvas = canvas
        self._ind = None
        self.background = None
        self.update_lines()

    def refresh(self):
        """
重新绘制所有的三角形
"""
        self.update_lines()
        self.canvas.draw()
        self.version += 1

    def del_triangle(self):
        """
删除最后一个三角形
"""
        self.points = self.points[:-3].copy()
        self.refresh()

    def add_triangle(self):
        """
添加一个三角形
"""
        self.points = np.vstack((self.points, np.array([(0,0),(1,0
        self.refresh()

    def set_points(self, points):
        """
直接设置三角形定点
"""
        self.points = points.copy()
        self.refresh()

    def get_eqs(self):
        """
计算所有的仿射方程
"""
        eqs = []
        for i in range(1,len(self.points)/3):
            eqs.append( solve_eq( self.points[:3,:], self.points[i
```

```
        return eqs

    def get_areas(self):
        """
通过三角形的面积计算仿射方程的迭代概率
"""
        areas = []
        for i in range(1, len(self.points)/3):
            areas.append( triangle_area(self.points[i*3:i*3+3,:]) )
        s = sum(areas)
        return [x/s for x in areas]

    def update_lines(self):
        """
重新绘制所有的三角形
"""
        del self.ax.lines[:]
        for i in xrange(0,len(self.points),3):
            color = self.colors[i/3%len(self.colors)]
            x0, x1, x2 = self.points[i:i+3, 0]
            y0, y1, y2 = self.points[i:i+3, 1]
            type = color+"%so"
            if i==0:
                linewidth = 3
            else:
                linewidth = 1
            self.ax.plot([x0,x1],[y0,y1], type % "-", linewidth=lin
            self.ax.plot([x1,x2],[y1,y2], type % "--", linewidth=li
            self.ax.plot([x0,x2],[y0,y2], type % ":", linewidth=lin

        self.ax.set_ylim(-10,10)
        self.ax.set_xlim(-10,10)

    def button_release_callback(self, event):
        """
鼠标按键松开事件
"""
        self._ind = None

    def button_press_callback(self, event):
        """
鼠标按键按下事件
"""
        if event.inaxes!=self.ax: return
        if event.button != 1: return
        self._ind = self.get_ind_under_point(event.xdata, event.yda

    def get_ind_under_point(self, mx, my):
        """
找到距离mx, my最近的顶点
"""
        for i, p in enumerate(self.points):
            if abs(mx-p[0]) < 0.5 and abs(my-p[1])< 0.5:
```

```
                return i
        return None

    def motion_notify_callback(self, event):
        """
 鼠标移动事件
 """
        self.event = event
        if self._ind is None: return
        if event.inaxes != self.ax: return
        if event.button != 1: return
        x,y = event.xdata, event.ydata

        #更新定点坐标
        self.points[self._ind,:] = [x, y]

        i = self._ind / 3 * 3
        # 更新顶点对应的三角形线段
        x0, x1, x2 = self.points[i:i+3, 0]
        y0, y1, y2 = self.points[i:i+3, 1]
        self.ax.lines[i].set_data([x0,x1],[y0,y1])
        self.ax.lines[i+1].set_data([x1,x2],[y1,y2])
        self.ax.lines[i+2].set_data([x0,x2],[y0,y2])

        # 背景为空时, 捕捉背景
        if self.background == None:
            self.ax.clear()
            self.ax.set_axis_off()
            self.canvas.draw()
            self.background = self.canvas.copy_from_bbox(self.ax.bb
            self.update_lines()

        # 快速绘制所有三角形
        self.canvas.restore_region(self.background) #恢复背景
        # 绘制所有三角形
        for line in self.ax.lines:
            self.ax.draw_artist(line)
        self.canvas.blit(self.ax.bbox)

        self.version += 1

class AskName(HasTraits):
    name = Str("")
    view = View(
        Item("name", label = u"名称"),
        kind = "modal",
        buttons = OKCancelButtons
    )

class IFSHandler(Handler):
    """
 在界面显示之前需要初始化的内容
 """
```

```
    def init(self, info):
        info.object.init_gui_component()
        return True

class IFSDesigner(HasTraits):
    figure = Instance(Figure) # 控制绘图控件的Figure对象
    ifs_triangle = Instance(IFSTriangles)
    add_button = Button(u"添加三角形")
    del_button = Button(u"删除三角形")
    save_button = Button(u"保存当前IFS")
    unsave_button = Button(u"删除当前IFS")
    clear = Bool(True)
    exit = Bool(False)
    ifs_names = List()
    ifs_points = List()
    current_name = Str

    view = View(
        VGroup(
            HGroup(
                Item("add_button"),
                Item("del_button"),
                Item("current_name", editor = EnumEditor(name="obje
                Item("save_button"),
                Item("unsave_button"),
                show_labels = False
            ),
            Item("figure", editor=MPLFigureEditor(), show_label=Fal
        ),
        resizable = True,
        height = 350,
        width = 600,
        title = u"迭代函数系统设计器",
        handler = IFSHandler()
    )

    def _current_name_changed(self):
        self.ifs_triangle.set_points( self.ifs_points[ self.ifs_nar

    def _add_button_fired(self):
        """
 添加三角形按钮事件处理
 """
        self.ifs_triangle.add_triangle()

    def _del_button_fired(self):
        self.ifs_triangle.del_triangle()

    def _unsave_button_fired(self):
        if self.current_name in self.ifs_names:
            index = self.ifs_names.index(self.current_name)
            del self.ifs_names[index]
            del self.ifs_points[index]
```

```
            self.save_data()

    def _save_button_fired(self):
        """
保存按钮处理
"""
        ask = AskName(name = self.current_name)
        if ask.configure_traits():
            if ask.name not in self.ifs_names:
                self.ifs_names.append( ask.name )
                self.ifs_points.append( self.ifs_triangle.points.co
            else:
                index = self.ifs_names.index(ask.name)
                self.ifs_names[index] = ask.name
                self.ifs_points[index] = self.ifs_triangle.points.c
            self.save_data()

    def save_data(self):
        with file("IFS.data", "wb") as f:
            pickle.dump(self.ifs_names[:], f) # ifs_names不是list, [
            for data in self.ifs_points:
                np.save(f, data) # 保存多个数组

    def ifs_calculate(self):
        """
在别的线程中计算
"""
        def draw_points(x, y, c):
            if len(self.ax2.collections) < ITER_TIMES:
                try:
                    self.ax2.scatter(x, y, s=1, c=c, marker="s", l
                    self.ax2.set_axis_off()
                    self.ax2.axis("equal")
                    self.figure.canvas.draw()
                except:
                    pass

        def clear_points():
            self.ax2.clear()

        while 1:
            try:
                if self.exit == True:
                    break
                if self.clear == True:
                    self.clear = False
                    self.initpos = [0, 0]
                    # 不绘制迭代的初始100个点
                    x, y, c = ifs( self.ifs_triangle.get_areas(), s
                    self.initpos = [x[-1], y[-1]]
                    self.ax2.clear()

                x, y, c = ifs( self.ifs_triangle.get_areas(), self.
```

```
                if np.max(np.abs(x)) < 1000000 and np.max(np.abs(y
                    self.initpos = [x[-1], y[-1]]
                    wx.CallAfter( draw_points, x, y, c )
                time.sleep(0.05)
            except:
                pass

    @on_trait_change("ifs_triangle.version")
    def on_ifs_version_changed(self):
        """
当三角形更新时，重新绘制所有的迭代点
"""
        self.clear = True

    def _figure_default(self):
        """
figure属性的缺省值，直接创建一个Figure对象
"""
        figure = Figure()
        self.ax = figure.add_subplot(121)
        self.ax2 = figure.add_subplot(122)
        self.ax2.set_axis_off()
        self.ax.set_axis_off()
        figure.subplots_adjust(left=0,right=1,bottom=0,top=1,wspace
        figure.patch.set_facecolor("w")
        return figure

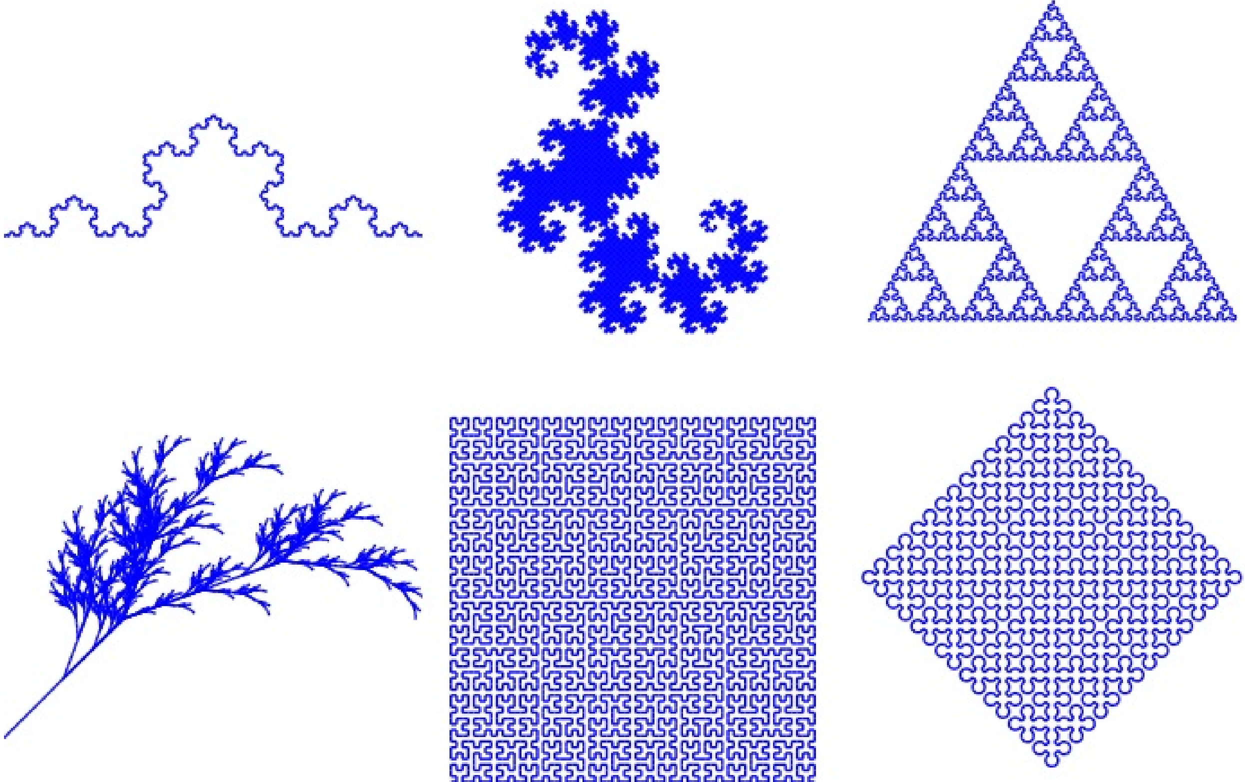
    def init_gui_component(self):
        self.ifs_triangle = IFSTriangles(self.ax)
        self.figure.canvas.draw()
        thread.start_new_thread( self.ifs_calculate, ())
        try:
            with file("ifs.data","rb") as f:
                self.ifs_names = pickle.load(f)
                self.ifs_points = []
                for i in xrange(len(self.ifs_names)):
                    self.ifs_points.append(np.load(f))

            if len(self.ifs_names) > 0:
                self.current_name = self.ifs_names[-1]
        except:
            pass

designer = IFSDesigner()
designer.configure_traits()
designer.exit = True
```

绘制L-System的分形图

相关文档： *L-System分形*



```
# -*- coding: utf-8 -*-
#L-System(Lindenmayer system)是一种用字符串替代产生分形图形的算法
from math import sin, cos, pi
import matplotlib.pyplot as pl
from matplotlib import collections

class L_System(object):
    def __init__(self, rule):
        info = rule['S']
        for i in range(rule['iter']):
            ninfo = []
            for c in info:
                if c in rule:
                    ninfo.append(rule[c])
                else:
                    ninfo.append(c)
            info = "".join(ninfo)
        self.rule = rule
        self.info = info

    def get_lines(self):
        d = self.rule['direct']
```

```
        a = self.rule['angle']
        p = (0.0, 0.0)
        l = 1.0
        lines = []
        stack = []
        for c in self.info:
            if c in "Ff":
                r = d * pi / 180
                t = p[0] + l*cos(r), p[1] + l*sin(r)
                lines.append(((p[0], p[1]), (t[0], t[1])))
                p = t
            elif c == "+":
                d += a
            elif c == "-":
                d -= a
            elif c == "[":
                stack.append((p,d))
            elif c == "]":
                p, d = stack[-1]
                del stack[-1]
        return lines

rules = [
    {
        "F":"F+F--F+F", "S":"F",
        "direct":180,
        "angle":60,
        "iter":5,
        "title":"Koch"
    },
    {
        "X":"X+YF+", "Y":"-FX-Y", "S":"FX",
        "direct":0,
        "angle":90,
        "iter":13,
        "title":"Dragon"
    },
    {
        "f":"F-f-F", "F":"f+F+f", "S":"f",
        "direct":0,
        "angle":60,
        "iter":7,
        "title":"Triangle"
    },
    {
        "X":"F-[[X]+X]+F[+FX]-X", "F":"FF", "S":"X",
        "direct":-45,
        "angle":25,
        "iter":6,
        "title":"Plant"
    },
    {
        "S":"X", "X":"-YF+XFX+FY-", "Y":"+XF-YFY-FX+",
```

```
        "direct":0,
        "angle":90,
        "iter":6,
        "title":"Hilbert"
    },
    {
        "S":"L--F--L--F", "L":"+R-F-R+", "R":"-L+F+L-",
        "direct":0,
        "angle":45,
        "iter":10,
        "title":"Sierpinski"
    },

]

def draw(ax, rule, iter=None):
    if iter!=None:
        rule["iter"] = iter
    lines = L_System(rule).get_lines()
    linecollections = collections.LineCollection(lines)
    ax.add_collection(linecollections, autolim=True)
    ax.axis("equal")
    ax.set_axis_off()
    ax.set_xlim(ax.dataLim.xmin, ax.dataLim.xmax)
    ax.invert_yaxis()

fig = pl.figure(figsize=(7,4.5))
fig.patch.set_facecolor("w")

for i in xrange(6):
    ax = fig.add_subplot(231+i)
    draw(ax, rules[i])

fig.subplots_adjust(left=0,right=1,bottom=0,top=1,wspace=0,hspace=0
pl.show()
```